

# **Teradata Vantage™ - SQL External Routine Programming**

---

Release 17.10

July 2021

# Copyright and Trademarks

Copyright © 2000 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

## Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

## Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

## Warranty Disclaimer

**Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.**

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

## Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

## Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: [docs@teradata.com](mailto:docs@teradata.com).

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

# Contents

<b>Chapter 1: Introduction to SQL External Routine Programming</b>	<b>14</b>
Changes and Additions	14
<b>Chapter 2: C/C++ User-Defined Functions</b>	<b>15</b>
UDF Types	15
Overall Procedure Synopsis	17
UDF Source Code Development	20
Header Files	24
C/C++ Function Name	26
Function Parameters	26
Function Name Overloading	29
Parameter Types in Overloaded Functions	32
Scalar Function Parameter List	34
Scalar Function Body	38
Aggregate Function Parameter List	40
Aggregate Function Body	46
Window Aggregate Function Parameter List	48
Window Aggregate Function Body	55
Intermediate Aggregate Storage	58
Table Function Parameter List	63
Constant Mode Table Function Body	70
Variable Mode Table Function Body	76
Table Operators	86
Returning SQLSTATE Values	103
Using Standard C Library Functions	106
Problems Using System V IPC and POSIX IPC	107
Installing the Function	108
Debugging a User-Defined Function	112
Resolving UDF Server Setup Errors	118
UDF Invocation	119
UDF Locations	122
Overloaded Function Invocation	123
Protected Mode Function Execution	128
Argument and Result Behavior	129
Data Type Conversion of Arguments	131
Defining Functions that Use LOB Types	133
Defining Functions that Use UDT Types	138
Defining Functions that Use Period Types	144
Defining Functions that Use the TD_ANYTYPE Type	148

Defining Functions that Use ARRAY Types . . . . .	150
Defining Functions for Algorithmic Compression . . . . .	155
Defining Functions for Row Level Security . . . . .	161
General Global Functions . . . . .	170
<b>Chapter 3: C/C++ External Stored Procedures . . . . .</b>	<b>174</b>
Overall Development Synopsis . . . . .	174
Source Code Development for C/C++ External Stored Procedures . . . . .	178
C/C++ Header Files . . . . .	182
C/C++ Function Name . . . . .	184
External Stored Procedure Parameters . . . . .	184
External Stored Procedure Parameter List . . . . .	185
External Stored Procedure C/C++ Function Body . . . . .	190
Returning SQLSTATE Values . . . . .	193
Calling Stored Procedures From External Stored Procedures . . . . .	195
Executing SQL in C/C++ External Stored Procedures . . . . .	200
Installing a C/C++ External Stored Procedure . . . . .	207
Debugging an External Stored Procedure . . . . .	214
Resolving UDF Server Setup Errors . . . . .	215
External Stored Procedure Invocation . . . . .	215
Protected Mode Execution . . . . .	216
AT TIME ZONE Option for External Procedures . . . . .	217
Argument Behavior . . . . .	218
<b>Chapter 4: C/C++ User-Defined Methods . . . . .</b>	<b>221</b>
UDM Types . . . . .	221
Overall Development Synopsis . . . . .	222
UDM Source Code Development . . . . .	223
Header Files . . . . .	225
C/C++ Function Name . . . . .	227
UDM Parameters . . . . .	227
Method Name Overloading . . . . .	228
UDM Parameter List . . . . .	229
C/C++ Function Body . . . . .	234
Returning SQLSTATE Values . . . . .	238
Installing the UDM . . . . .	240
Debugging a UDM . . . . .	245
Resolving UDF Server Setup Errors . . . . .	246
UDM Invocation . . . . .	246
Protected Mode Execution . . . . .	248
Argument Behavior . . . . .	249
<b>Chapter 5: Java User-Defined Functions . . . . .</b>	<b>253</b>
UDF Types . . . . .	253
Java Development Environment . . . . .	253

Overall Procedure Synopsis . . . . .	254
UDF Source Code Development . . . . .	255
Class and Method Names . . . . .	258
Parameter List and Return Value . . . . .	258
Scalar UDFs . . . . .	261
Aggregate UDFs . . . . .	264
Window Aggregate UDFs . . . . .	268
Intermediate Aggregate Storage . . . . .	271
Table UDFs . . . . .	276
Constant Mode Table UDFs . . . . .	281
Variable Mode Table UDFs . . . . .	287
Table UDFs that Retain Data Between Iterations . . . . .	299
Table Operators . . . . .	307
Exception Handling . . . . .	316
Compiling the Source Code . . . . .	318
Registering the JAR or ZIP File . . . . .	319
Defining the SQL Function . . . . .	319
Debugging a User-Defined Function . . . . .	320
Resolving UDF Server Setup Errors . . . . .	321
UDF Invocation . . . . .	321
Java UDF Execution Environment . . . . .	324
Function Name Overloading . . . . .	324
Defining Functions that Use the TD_ANYTYPE Type . . . . .	326
JAR and ZIP File Administration . . . . .	327
Using Java Reflection . . . . .	328
<b>Chapter 6: Java External Stored Procedures . . . . .</b>	<b>330</b>
Java Development Environment . . . . .	330
Overall Development Synopsis . . . . .	330
Class and Method Names . . . . .	332
Parameter List . . . . .	332
External Stored Procedures That Use TD_ANYTYPE Arguments . . . . .	337
Class Fields . . . . .	338
Returning SQLSTATE Values . . . . .	338
Executing SQL in Java External Stored Procedures . . . . .	338
Returning Dynamic Result Sets . . . . .	342
Consuming Result Sets Created by Calling a Stored Procedure . . . . .	344
Teradata Application Classes . . . . .	345
Compiling the Source Code . . . . .	345
Registering the JAR or ZIP File . . . . .	346
Defining the SQL External Stored Procedure . . . . .	347
Debugging Using Trace Tables . . . . .	348
Resolving UDF Server Setup Errors . . . . .	350
External Stored Procedure Invocation . . . . .	351
AT TIME ZONE Option for External Procedures . . . . .	351

Argument Behavior .....	352
JAR and ZIP File Administration .....	353
Using Java Reflection .....	354
Attempting to Exit the Java Virtual Machine .....	354
<b>Chapter 7: R Table Operators .....</b>	<b>355</b>
Installation of R Components and Packages .....	355
Using ExecR to Execute R Scripts and Table Operators .....	356
Supported Character Sets for R Programs .....	357
Memory Limitation .....	358
keepLog USING Clause .....	359
R Table Operator Example .....	359
R Table Operator Example: Echo Example .....	360
R Table Operator Use Case: Grouping Using K Means .....	362
R FNC Functions .....	371
Mapping Between R and C Data Types .....	376
Mapping Between Teradata Data Types and R Types .....	377
tdr.ampinfo .....	378
tdr.BigInt .....	379
tdr.Blob .....	379
tdr.Byte .....	379
tdr.ByteInt .....	380
tdr.Char .....	380
tdr.Clob .....	381
tdr.Close .....	381
tdr.Date .....	382
tdr.dbsinfo .....	382
tdr.Decimal1 .....	383
tdr.Decimal2 .....	383
tdr.Decimal4 .....	384
tdr.Decimal8 .....	385
tdr.Decimal16 .....	385
tdr.DisableCoGroup .....	386
tdr.Float .....	386
tdr.GetAsClauseName .....	387
tdr.GetAttributeByNdx .....	387
tdr.GetColCount .....	390
tdr.GetColDef .....	390
tdr.GetContractDef .....	392
tdr.GetContractLength .....	392
tdr.GetCountHashBy .....	393
tdr.GetCountLocalOrderBy .....	393
tdr.GetCustomKeyAt .....	394
tdr.GetCustomKeyCount .....	394
tdr.GetCustomValuesAt .....	394

tdr.GetCustomValuesOf	395
tdr.GetFormat	395
tdr.GetHashByDef	396
tdr.GetLobLength	397
tdr.GetLocalOrderByDef	398
tdr.getnodedata	399
tdr.GetStreamCount	399
tdr.Integer	399
tdr.IntervalDay	400
tdr.IntervalDTH	400
tdr.IntervalDTM	400
tdr.IntervalDTS	401
tdr.IntervalHour	401
tdr.IntervalHTM	401
tdr.IntervalHTS	402
tdr.IntervalMinute	402
tdr.IntervalMonth	402
tdr.IntervalMTS	403
tdr.IntervalSecond	403
tdr.IntervalYear	403
tdr.IntervalYTM	404
tdr.IsDimension	404
tdr.LobAppend	404
tdr.LobClose	405
tdr.LobCol2Loc	407
tdr.LobOpen_CL	407
tdr.LobRead	409
tdr.Open	410
tdr.Read	411
tdr.Real	413
tdr.SetAttributeByNdx	413
tdr.SetContractDef	415
tdr.SetError	415
tdr.SetFormat	416
tdr.SetHashByDef	417
tdr.SetLocalOrderByDef	418
tdr.SetOutputColDef	419
tdr.SmallInt	420
tdr.TblRead	420
tdr.TblWrite	422
tdr.Time	423
tdr.Timestamp	424
tdr.TimestampWTZ	424
tdr.TimeWTZ	424
tdr.tracestring	425

tdr.tracewrite . . . . .	425
tdr.VarByte . . . . .	426
tdr.VarChar . . . . .	426
tdr.Write . . . . .	427
<b>Chapter 8: Global and Persistent Data . . . . .</b>	<b>429</b>
About GLOP Data . . . . .	429
Benefits of GLOP Data . . . . .	430
GLOP Mappings . . . . .	430
GLOP Set . . . . .	431
GLOP Set Membership . . . . .	432
GLOP Data Access From an External Routine . . . . .	432
GLOP Page . . . . .	434
GLOP Types . . . . .	435
Persistence of GLOP Types . . . . .	438
GLOP Data Attributes . . . . .	438
GLOP Data System Tables . . . . .	440
GLOP Data System Stored Procedures . . . . .	440
Managing GLOP Data . . . . .	441
Creating a GLOP Set Definition . . . . .	441
Adding Mappings and Data to a GLOP Set . . . . .	441
Becoming a Member of a GLOP Set . . . . .	442
Using GLOP Data in a C/C++ External Routine . . . . .	442
DBCExtension.GLOP_Add Stored Procedure . . . . .	444
DBCExtension.GLOP_Remove Stored Procedure . . . . .	450
DBCExtension.GLOP_Change Stored Procedure . . . . .	453
DBCExtension.GLOP_Report Stored Procedure . . . . .	457
System Disk Space . . . . .	460
Global Configuration Settings . . . . .	460
<b>Chapter 9: Administration . . . . .</b>	<b>461</b>
System Requirements . . . . .	461
Global Configuration Settings . . . . .	461
Protected Mode Process and Server Administration for C/C++ External Routines . . . . .	462
Server Administration for Java External Routines . . . . .	462
Finding Unprotected Mode UDFs and Changing the Execution Mode . . . . .	463
File System Cleanup . . . . .	465
Registering and Distributing JAR and ZIP Files for Java External Routines . . . . .	465
Removing Registered JAR or ZIP Files . . . . .	469
Replacing Registered JAR or ZIP Files . . . . .	470
Redistributing Registered JAR or ZIP Files . . . . .	472
Altering the Java Path of Registered JAR or ZIP Files . . . . .	473
Distributing Packages . . . . .	475
Backing Up and Restoring Packages [Deprecated] . . . . .	480



<b>Chapter 10: SQL Data Type Mapping</b>	<b>489</b>
C Data Types	489
Java Data Types	517
<b>Chapter 11: C Library Functions</b>	<b>537</b>
Function Types	537
FNC Data Structures	550
FNC_AmplInfo	556
FNC_CallSP	558
FNC_CheckNullBitVector	563
FNC_CheckNullBitVectorByElemIndex	565
FNC_DbsInfo [Deprecated]	568
FNC_DbsInfo_EON	569
FNC_DBSSessionAttrInfo	570
FNC_DefMem	572
FNC_free	573
FNC_GeomGetResultWKBBlob	574
FNC_GeomGetResultWKTClob	576
FNC_GeomGetWKB	579
FNC_GeomGetWKBBlob	581
FNC_GeomGetWKBSize	583
FNC_GeomGetWKT	584
FNC_GeomGetWKTClob	587
FNC_GeomGetWKTSize	589
FNC_GeomSetWKB	591
FNC_GeomSetWKBBlob	593
FNC_GeomSetWKT	595
FNC_GeomSetWKTClob	597
FNC_GetAmphHash	600
FNC_GetAnyTypeParamInfo [Deprecated]	601
FNC_GetAnyTypeParamInfo_eon	606
FNC_GetArrayElementCount	610
FNC_GetArrayElements	612
FNC_GetArrayNumDimensions	616
FNC_GetArrayTypeInfo [Deprecated]	617
FNC_GetArrayTypeInfo_EON	621
FNC_GetByteLength	625
FNC_GetCharLength	626
FNC_GetDatasetInfo	627
FNC_GetDatasetInputLob	630
FNC_GetDatasetResultLob	633
FNC_GetDatasetSchema	635
FNC_GetDatasetSchemaLob	639
FNC_GetDistinctInputLob	641

FNC_GetDistinctResultLob . . . . .	642
FNC_GetDistinctValue . . . . .	644
FNC_GetExtendedJSONInfo . . . . .	647
FNC_GetGeometryInfo . . . . .	649
FNC_Get_GLOP_Map . . . . .	650
FNC_GetGraphicLength . . . . .	654
FNC_GetHashAmp . . . . .	655
FNC_GetInternalValue . . . . .	656
FNC_GetJSONInfo . . . . .	660
FNC_GetJSONInputLob . . . . .	663
FNC_GetJSONResultLob . . . . .	665
FNC_GetLobLength . . . . .	668
FNC_GetLobLength_CL . . . . .	669
FNC_GetOutputBufferSize . . . . .	670
FNC_GetPhase . . . . .	670
FNC_GetPhaseEx . . . . .	674
FNC_GetQueryBand . . . . .	680
FNC_GetQueryBandU . . . . .	683
FNC_GetQueryBandPairs . . . . .	686
FNC_GetQueryBandPairsU . . . . .	688
FNC_GetQueryBandValue . . . . .	694
FNC_GetQueryBandValueU . . . . .	696
FNC_GetStructuredAttribute . . . . .	701
FNC_GetStructuredAttributeByNdx . . . . .	705
FNC_GetStructuredAttributeCount . . . . .	709
FNC_GetStructuredAttributeInfo [Deprecated] . . . . .	710
FNC_GetStructuredAttributeInfo_EON . . . . .	715
FNC_GetStructuredInputLobAttribute . . . . .	719
FNC_GetStructuredInputLobAttributeByNdx . . . . .	721
FNC_GetStructuredResultLobAttribute . . . . .	723
FNC_GetStructuredResultLobAttributeByNdx . . . . .	725
FNC_GetUDTHandles . . . . .	727
FNC_GetVarCharLength . . . . .	729
FNC_GetXML . . . . .	730
FNC_GetXMLBlob . . . . .	731
FNC_GetXMLByte . . . . .	733
FNC_GetXMLClob . . . . .	734
FNC_GetXMLInfo . . . . .	736
FNC_GetXMLResultBlob . . . . .	738
FNC_GetXMLResultClob . . . . .	739
FNC_GLOP_Global_Copy . . . . .	739
FNC_GLOP_Lock . . . . .	741
FNC_GLOP_Map_Page . . . . .	743
FNC_GLOP_Unlock . . . . .	745
FNC_LobAppend . . . . .	746

FNC_LobClose .....	748
FNC_LobCol2Loc .....	750
FNC_LobLoc2Ref .....	750
FNC_LobOpen .....	752
FNC_LobOpen_CL .....	754
FNC_LobRead .....	756
FNC_LobRef2Loc .....	758
FNC_MBBGetValue .....	760
FNC_MBBSetValue .....	762
FNC_MBRGetValue .....	764
FNC_MBRSetValue .....	766
FNC_malloc .....	767
FNC_SetActivityCount .....	769
FNC_SetArrayElements .....	769
FNC_SetArrayElementsWithMultiValues .....	775
FNC_SetDatasetLob .....	780
FNC_SetDistinctValue .....	783
FNC_SetInternalValue .....	786
FNC_SetNullBitVector .....	790
FNC_SetNullBitVectorByElemIndex .....	792
FNC_SetStructuredAttribute .....	794
FNC_SetStructuredAttributeByNdx .....	798
FNC_SetVarCharLength .....	801
FNC_SetXML .....	802
FNC_SetXMLBlob .....	804
FNC_SetXMLByte .....	806
FNC_SetXMLClob .....	808
FNC_TblAbort .....	810
FNC_TblAllocCtrlCtx .....	812
FNC_TblAllocCtx .....	814
FNC_TblControl .....	816
FNC_TblFirstParticipant .....	817
FNC_TblGetColDef .....	819
FNC_TblGetCtrlCtx .....	822
FNC_TblGetCtx .....	824
FNC_TblGetNodeData .....	825
FNC_TblOpBindAttributeByNdx .....	827
FNC_TblOpBytesTransferred .....	829
FNC_TblOpClose .....	829
FNC_TblOpDisableCoGroup .....	830
FNC_TblOpGetAsClauseName .....	831
FNC_TblOpGetAttributeByNdx .....	831
FNC_TblOpGetBaseInfo .....	833
FNC_TblOpGetColCount .....	836
FNC_TblOpGetColDef .....	836

FNC_TblOpGetContractDef . . . . .	838
FNC_TblOpGetContractLength . . . . .	838
FNC_TblOpGetContractPhase . . . . .	839
FNC_TblOpGetCountHashByDef . . . . .	839
FNC_TblOpGetCountLocalOrderByDef . . . . .	840
FNC_TblOpGetCustomKeyCount . . . . .	840
FNC_TblOpGetCustomKeyInfoAt . . . . .	841
FNC_TblOpGetCustomKeyInfoOf . . . . .	842
FNC_TblOpGetCustomValuesOf . . . . .	843
FNC_TblOpGetExternalQuery . . . . .	843
FNC_TblOpGetFormat . . . . .	846
FNC_TblOpGetFunctionDef . . . . .	847
FNC_TblOpGetHashByDef . . . . .	848
FNC_TblOpGetInnerContract . . . . .	848
FNC_TblOpGetLocalOrderByDef . . . . .	849
FNC_TblOpGetStreamCount . . . . .	850
FNC_TblOpGetStructuredAttributeInfo . . . . .	850
FNC_TblOpGetUDTMetadata . . . . .	855
FNC_TblOpGetUniqID . . . . .	860
FNC_TblOpIsDimension . . . . .	860
FNC_TblOpOpen . . . . .	861
FNC_TblOpRead . . . . .	862
FNC_TblOpReadBuf . . . . .	863
FNC_TblOpReadBufEx . . . . .	864
FNC_TblOpSetContractDef . . . . .	865
FNC_TblOpSetDisplayLength . . . . .	865
FNC_TblOpSetError . . . . .	866
FNC_TblOpSetExplainText . . . . .	867
FNC_TblOpSetFormat . . . . .	868
FNC_TblOpSetHashByDef . . . . .	871
FNC_TblOpSetInputColTypes . . . . .	872
FNC_TblOpSetLocalOrderByDef . . . . .	872
FNC_TblOpSetOutputColDef . . . . .	873
FNC_TblOpWrite . . . . .	874
FNC_TblOpWriteBuf . . . . .	875
FNC_TblOptOut . . . . .	875
FNC_Trace_String . . . . .	877
FNC_Trace_Write . . . . .	877
FNC_Trace_Write_DL . . . . .	880
FNC_UdtDeserialize . . . . .	883
FNC_UdtGetSerializeSize . . . . .	884
FNC_UdtSerialize . . . . .	885
FNC_UdtSerializeSupported . . . . .	885
FNC_Where_Am_I . . . . .	886

<b>Chapter 12: Java Application Classes</b>	<b>889</b>
Class Path	890
com.teradata.fnc.AMPInfo	890
com.teradata.fnc.Array	891
com.teradata.fnc.Blob	909
com.teradata.fnc.Clob	917
com.teradata.fnc.Context	927
com.teradata.fnc.DbsInfo	933
com.teradata.fnc.NodeInfo	935
com.teradata.fnc.Operator	937
com.teradata.fnc.operator.Metadata	938
com.teradata.fnc.Phase	939
com.teradata.fnc.QueryBand	940
com.teradata.fnc.Runtime	945
com.teradata.fnc.SQLXML	987
com.teradata.fnc.ST_Geometry	993
com.teradata.fnc.Struct	998
com.teradata.fnc.TeradataType	1001
com.teradata.fnc.Tbl	1002
com.teradata.fnc.value.TeradataTime	1016
com.teradata.fnc.value.TeradataTimestamp	1017
com.teradata.fnc.TraceObj	1018
<b>Appendix A: UDF Code Examples</b>	<b>1024</b>
<b>Appendix B: External Stored Procedure Code Examples</b>	<b>1138</b>
<b>Appendix C: UDM Code Examples</b>	<b>1171</b>
<b>Appendix D: C/C++ Command-line Debugging for UDFs</b>	<b>1184</b>
<b>Appendix E: Procedure to Enable R Functionality with ExecR</b>	<b>1207</b>
<b>Appendix F: Additional Information</b>	<b>1214</b>

# Introduction to SQL External Routine Programming

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

*Teradata Vantage™ - SQL External Routine Programming* describes how to program Teradata user-defined functions (UDFs), user-defined methods (UDMs), and external stored procedures, which are collectively called external routines.

---

**Note:**

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

---

## Changes and Additions

Date	Description
July 2021	Minor edits.
June 2020	New SET SESSION UDFSEARCHPATH statement can be used to set a preferred search path for UDF execution. See <a href="#">UDF Locations</a> .
	Customers using Vantage delivered as-a-service (Vantage running on AWS or Azure), cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

# C/C++ User-Defined Functions

Teradata provides a wide range of built-in functions that address many of the needs of typical users. In those cases when standard functions are inadequate, Teradata allows you to define user-defined functions (UDFs) that extend the capability of normal SQL within the database.

UDFs are database objects that you build or acquire. For example, you can install UDF objects or packages from third party vendors without providing the source code. A UDF is similar to standard SQL functions such as SQRT, ABS, or TRIM and is invoked in the same manner.

You can write UDFs in C or C++, then compile them into shared objects. Once compiled, you can use the UDF in SQL statements for activities such as enforcing business rules or aiding in the transformation of data.

UDFs execute in parallel within the database. However, the developer of the function can direct which AMPs (virtualized parallel processors) will participate and which AMPs will not. UDFs execute under the control of the AMP and can be very efficient doing row-by-row complex analyses. They are scalable and inherit all of the natural parallelism in Teradata.

You can use UDFs for external I/O, almost any type of data conversion, encapsulating complex business calculations, complex statistical calculations using the aggregate function class, and a wide range of other uses.

Additional details are available at <https://www.teradata.com>.

For details on writing your own functions in the Java programming language, see [Java User-Defined Functions](#).

## UDF Types

Teradata supports three types of UDFs:

- Scalar
- Aggregate and Window Aggregate
- Table Functions and Operators

		Output Parameter Type	
		Scalar	Set
Input Parameter Type	Scalar	User Defined Scalar Function	User Defined Table Function
	Set	User Defined Aggregate Function	User Defined Table Operator

## Scalar Functions

Scalar functions take input arguments and return a single value result. Some examples of standard SQL scalar functions are CHARACTER\_LENGTH, POSITION, and SUBSTRING.

You can use a scalar function in place of a column name in an expression. When Vantage evaluates the expression, it invokes the scalar function. No context is retained after the function completes.

You can also use a scalar function to implement user-defined type (UDT) functionality such as cast, transform, or ordering.

## Aggregate Functions

Aggregate functions produce summary results. They differ from scalar functions in that they take grouped sets of relational data, make a pass over each group, and return one result for the group. Some examples of standard SQL aggregate functions are AVG, SUM, MAX, and MIN.

Vantage invokes an aggregate function once for each item in the group, passing the detail values of a group through the input arguments. To accumulate summary information, an aggregate function must retain context each time it is called.

You do not need to understand or worry about how to create a group, or how to code an aggregate UDF to deal with groups. Vantage automatically takes care of all of those difficult aspects. You only need to write the basic algorithm of combining the data passed in to produce the desired result.

## Window Aggregate Functions

You can apply the ordered analytical window feature to a user-defined aggregate function.

Ordered analytical functions provide support for common operations in analytical processing that require an ordered set of rows or use the values from multiple rows in computing a new value.

The window feature provides a way to dynamically define a subset of data, or window, and allows the aggregate function to operate on that window of rows. Without a window specification, aggregate functions return one value for all qualified rows examined, but window aggregate functions return a new value for each of the qualifying rows participating in the query.

## Table Functions

A table function is invoked in the FROM clause of an SQL SELECT statement and returns a table, a row at a time in a loop to the SELECT statement. The function can produce the rows of a table from the input arguments passed to it or by reading an external file or message queue.

The number of columns in the rows that a table function returns can be specified dynamically at runtime in the SELECT statement that invokes the table function.

---

### Note:

Table functions and table operators cannot execute against fallback data when an AMP is down. Once the AMP returns to service, the query can be submitted.

---



## Table Operators

A table operator is invoked in the FROM clause of an SQL SELECT statement and accepts one or more tables or table expressions as input and generates a table as output. For more information, see [Table Operators](#).

## Overall Procedure Synopsis

The steps that you take to develop, compile, install, and use a UDF depend on whether the UDF issues operating system I/O calls and whether the UDF implements UDT functionality, such as transform or ordering. The meaning of I/O as it applies to UDFs is any operating system call that requires the operating system to retain a resource context, such as for open files or other operating system services. Such resource usage usually returns a handle to the caller that is used to access the resource and must be released when finished using it.

## UDFs that Issue Operating System I/O Calls

Here is a synopsis of the steps you take to develop, compile, install, and use a UDF that issues operating system I/O calls:

1. Determine the level of access to operating system services that the UDF requires.

IF the UDF ...	THEN ...
accesses operating system resources that ordinary operating system users have access to	the UDF can run in protected execution mode as a separate process under 'tdatuser', a local operating system user that the database installation process creates.
requires access to specific operating system resources	use CREATE AUTHORIZATION or REPLACE AUTHORIZATION to create a context that identifies a native operating system user and allows UDFs to perform I/O by running as separate processes under the authorization of that user.

2. Use the CREATE FUNCTION or REPLACE FUNCTION statement to identify the location of the source code, object, or package, and install it on a development or test database.

**Recommendation:** In general, you should not create UDFs in Teradata system databases such as SYSLIB or SYSUDTLIB. For more information, see [Installing the Function](#).

IF you ...	THEN the CREATE FUNCTION or REPLACE FUNCTION statement ...
did not use CREATE AUTHORIZATION or REPLACE AUTHORIZATION in the previous step	sets the default execution mode for the UDF to protected mode. The UDF runs under the tdatuser operating system user and can access the system resources for which tdatuser has privileges.

IF you ...	THEN the CREATE FUNCTION or REPLACE FUNCTION statement ...
used CREATE AUTHORIZATION or REPLACE AUTHORIZATION in the previous step	must specify the EXTERNAL SECURITY clause to associate execution of the UDF with the context created by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement. The UDF runs under the operating system user identified by the specified context and can access the system resources for which the user has privileges.

The function is compiled, if the source code is submitted, linked to the dynamic linked library (DLL or SO) associated with the database in which the function resides, and distributed to all database nodes in the system.

3. Test and debug the UDF until you are satisfied it works correctly.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions. For more information, see [C/C++ Command-line Debugging for UDFs](#).

4. Install the UDF on your production database.
5. Use GRANT to grant privileges to users who are authorized to use the UDF.

## UDFs that Implement UDT Functionality

Here is a synopsis of the steps you take to develop, compile, install, and use a UDF that implements UDT transform, ordering, or cast functionality:

1. Write, test, and debug the C or C++ code for the UDF.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions for the database. For more information, see [C/C++ Command-line Debugging for UDFs](#).

2. Use the CREATE FUNCTION or REPLACE FUNCTION statement to identify the location of the source code or object, and install it on the server.

**Recommendation:** In general, you should not create UDFs in Teradata system databases such as SYSLIB or SYSUDTLIB. For more information, see [Installing the Function](#).

The function is compiled, if the source code is submitted, linked to the dynamic linked library (DLL or SO) associated with the database in which the function resides, and distributed to all database nodes in the system.

3. Register the UDF as a transform, ordering, or cast routine.

IF the UDF implements this functionality for a UDT ...	THEN use this statement to register the UDF ...
cast	CREATE CAST or REPLACE CAST
ordering	CREATE ORDERING or REPLACE ORDERING

IF the UDF implements this functionality for a UDT ...	THEN use this statement to register the UDF ...
transform	CREATE TRANSFORM or REPLACE TRANSFORM

4. Test the UDF in protected execution mode until you are satisfied it works correctly.

Protected mode is the default execution mode for a UDF. In protected mode, Vantage isolates all of the data the UDF might access as a separate process in its own local workspace. This makes the function run slower. If any memory violation or other system error occurs, the error is localized to the function and the transaction executing the function.

5. Use ALTER FUNCTION to change the UDF to run in nonprotected execution mode.

---

**Note:**

If you subsequently use REPLACE FUNCTION to replace the function, the execution mode reverts back to protected mode.

---

6. Rerun the tests from Step 4 to test the UDF in nonprotected execution mode until you are satisfied it works correctly.
7. Use GRANT to grant privileges to users who are authorized to use the UDF and the UDT.

## UDFs that Do Not Perform I/O and Do Not Implement UDT Functionality

Here is a synopsis of the steps you take to develop, compile, install, and use a UDF that does not perform I/O:

1. Use CREATE FUNCTION or REPLACE FUNCTION to identify the location of the source code, object, or package, and install it on a development or test database.

**Recommendation:** In general, you should not create UDFs in Teradata system databases such as SYSLIB or SYSUDTLIB. For more information, see [Installing the Function](#).

The function is compiled, if the source code is submitted, linked to the dynamic linked library (DLL or SO) associated with the database in which the function resides, and distributed to all database nodes in the system.

2. Test the UDF in *protected* execution mode until you are satisfied it works correctly.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions. For more information, see [C/C++ Command-line Debugging for UDFs](#).

Protected mode is the default execution mode for a UDF. In protected mode, Vantage isolates all of the data the UDF might access as a separate process in its own local workspace. This makes the function run slower. If any memory violation or other system error occurs, the error is localized to the function and the transaction executing the function.

3. Use ALTER FUNCTION to change the UDF to run in nonprotected execution mode.

**Note:**

If you subsequently use REPLACE FUNCTION to replace the function, the execution mode reverts back to protected mode.

4. Rerun the tests from Step 3 to test the UDF in nonprotected execution mode until you are satisfied it works correctly.
5. Install the UDF on your production database.
6. Use GRANT to grant privileges to users who are authorized to use the UDF.

## Related Information

FOR more information on ...	SEE ...
creating a UDF	related topics in this document.
debugging a UDF	<ul style="list-style-type: none"> <li>• <a href="#">Debugging a User-Defined Function.</a></li> <li>• <a href="#">Debugging Using Trace Tables.</a></li> <li>• <a href="#">C/C++ Command-line Debugging for UDFs.</a></li> </ul>
<ul style="list-style-type: none"> <li>• protected mode function execution</li> <li>• tdatuser operating system user</li> </ul>	<a href="#">Protected Mode Function Execution.</a>
code examples for scalar, aggregate, and table UDFs	<a href="#">UDF Code Examples.</a>
<ul style="list-style-type: none"> <li>• CREATE FUNCTION</li> <li>• REPLACE FUNCTION</li> </ul>	<ul style="list-style-type: none"> <li>• <a href="#">Installing the Function.</a></li> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
<ul style="list-style-type: none"> <li>• CREATE AUTHORIZATION</li> <li>• REPLACE AUTHORIZATION</li> </ul>	<ul style="list-style-type: none"> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
<ul style="list-style-type: none"> <li>• CREATE CAST</li> <li>• REPLACE CAST</li> <li>• CREATE ORDERING</li> <li>• REPLACE ORDERING</li> <li>• CREATE TRANSFORM</li> <li>• REPLACE TRANSFORM</li> </ul>	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

## UDF Source Code Development

You develop the body of a UDF using the C or C++ programming language.

### Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

## Source Code Contents

The topics that follow provide details about the contents of the source code for a UDF. In general, the contents of the source code must:

- Define the SQL\_TEXT constant
- Include the sqltypes\_td.h header file
- Define the function parameter list in the order that matches the parameter passing convention specified in the CREATE FUNCTION statement

### Note:

A C++ UDF or external procedure must catch all C++ exceptions by the end of the UDF or external procedure code. Any exception that is not caught and is thrown out from the routine may result in unexpected behavior.

## Internal and External Data

The following rules apply to internal and external data.

IF the UDF is ...	THEN the function ...
a scalar function	cannot retain global data from function call to function call unless it uses the Global and Persistent (GLOP) data feature. For more information, see <a href="#">Global and Persistent Data</a> .
an aggregate function	can use an intermediate storage area to keep partially aggregated data.
a table function	can use scratch pad memory keep track of its progress.

A function *cannot* contain any static variables. A function can contain static constants.

You can specify the External Data Access clause in the CREATE FUNCTION/REPLACE FUNCTION statement to define the relationship between a function or external stored procedure and data that is external to Vantage.

The option you specify determines:

- Whether or not the external routine can read or modify external data

- Whether or not the database will redrive the request involving the function or procedure after a database restart

If Redrive protection is enabled, the system preserves responses for completed SQL requests and resubmits uncompleted requests when there is a database restart. However, if the External Data Access clause of an external routine is defined with the MODIFIES EXTERNAL DATA option, then the database will not redrive the request involving that function or procedure. For details about Redrive functionality, see:

- Teradata Vantage™ - Database Administration*, B035-1093
- The RedriveProtection and RedriveDefaultParticipation DBS Control fields in *Teradata Vantage™ - Database Utilities*, B035-1102.

If you do not specify an External Data Access clause, the default is NO EXTERNAL DATA.

The following table explains the options for the External Data Access clause and how the database uses them for external routines.

Option	Description
MODIFIES EXTERNAL DATA	The routine modifies data that is external to the database. In this case, the word <i>modify</i> includes delete, insert, and update operations.  <b>Note:</b> Following a database failure, the database does not redrive requests involving a function or external stored procedure defined with this option.
NO EXTERNAL DATA	The routine does not access data that is external to the database. This is the default.
READS EXTERNAL DATA	The routine reads, but does not modify data that is external to the database.

For more information, see the information about CREATE FUNCTION (External Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## I/O

The following rules apply to I/O.

IF the UDF ...	THEN the function ...
runs in nonprotected execution mode	cannot do any I/O, including standard I/O and network I/O.
specifies the EXTERNAL SECURITY clause in the CREATE FUNCTION or REPLACE FUNCTION statement	can do I/O or otherwise interface with the operating system such that the operating system retains resources such as file handles or object handles. The function executes as a separate process under the authorization of a specific native operating system user established by a CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement. The function must release opened resources (close handles) before it completes. Table functions must release opened resources before completing the end phase.

IF the UDF ...	THEN the function ...
	All other types of functions must release opened resources before each instance of the function terminates.
runs in protected execution mode	<p>can do I/O or otherwise interface with the operating system such that the operating system retains resources such as file handles or object handles.</p> <p>The function executes as a separate process under the authorization of the 'tdatuser' operating system user.</p> <p>The function must release opened resources (close handles) before it completes. Table functions must release opened resources before completing the end phase. All other types of functions must release opened resources before each instance of the function terminates.</p> <p>For more information on protected execution mode, see <a href="#">Protected Mode Function Execution</a>.</p>

## Signal Handling

UDTs that run on Linux systems cannot use or modify the handling of the signal SIGUSR2, which is reserved by Teradata.

## Standard C Library Functions

A user-defined function that runs in nonprotected mode can call standard C library functions that do not do any I/O, such as string library functions.

Restrictions apply to some functions, such as *malloc* and *free*.

For more information, see [Using Standard C Library Functions](#).

## Teradata UDF Library Functions

Teradata provides functions that you can use for UDF development. For example, use the FNC\_DbsInfo function to get session information related to the current execution of a UDF.

The following table shows some of the categories of library functions that Teradata provides.

Category	Use
Aggregate Intermediate Storage	Allocates the intermediate storage you need for accumulating summary information in an aggregate function
ARRAY Data Type Interface	Enable a UDF to access and set the values of the elements in an ARRAY data type
Global Information	Returns session information about the current execution of a UDF
Lob Access	Enable a UDF to use a locator to access the contents of a referenced object and to append data to a LOB object

Category	Use
Period Data Type Interface	Enable a UDF to access and set the value of a Period data type
Query Band Access	Return query band name-value pairs that have been set on a session, transaction, or profile to identify the originating source of queries and help manage task priorities and track system use
String Argument and Result Processing	Useful for UDFs that are used for the algorithmic compression and decompression of table columns
Table Function Processing	Required to implement table functions
TD_ANYTYPE Parameter Access	Enable a UDF to retrieve information about TD_ANYTYPE input and output parameters
Trace	Let you get trace output for debugging purposes during UDF development
UDT Interface	Enable a UDF to access and set the value of a distinct UDT or attribute values of a structured UDT

For a list of all available functions and their descriptions, see [C Library Functions](#).

## Consequences of Using the exit() System Call

Avoid calling `exit()` from UDFs.

UDFs that call `exit()` generate the following results:

- The request that called the UDF is rolled back.
- The server process is terminated and must be recreated for the next UDF, causing additional overhead.

## Header Files

Teradata provides the `sqltypes_td.h` header file that you include in your source code file. The header file defines the equivalent C data types for all database data types that you can use for the input arguments and result of your UDFs. Every SQL data type corresponds to a C data type in `sqltypes_td.h`.

## Location

The header file is in the `etc` directory of the Teradata software distribution:

```
/usr/tdbms/etc
```

To verify the path of the `etc` directory, enter the following on the command line:

```
pdepath -e
```



## SQL\_TEXT Definition

Before you include the `sqltypes_td.h` header file, you must define the `SQL_TEXT` constant. The value that you use must match the current server character set of the session in which you use the `CREATE FUNCTION` statement to create the UDF.

IF you use the <code>CREATE FUNCTION</code> statement when the current server character set is ...	THEN the function must set <code>SQL_TEXT</code> to ...
KANJI	Kanji1_Text
KANJISJIS	Kanjisjis_Text
LATIN	Latin_Text
UNICODE	Unicode_Text

`SQL_TEXT` is used for specific UDF arguments when the UDF uses parameter style SQL. For more information on parameter styles, see [Scalar Function Parameter List](#), [Aggregate Function Parameter List](#), and [Table Function Parameter List](#).

Vantage remembers the character set the UDF was created under so the UDF can translate `SQL_TEXT` input arguments to the expected text and translate text to `SQL_TEXT` output arguments, no matter who calls the UDF and what the current server character set is for the session.

### NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

## Example: Defining `SQL_TEXT` and Including `sqltypes_td.h`

The following example shows how to define `SQL_TEXT` and include the `sqltypes_td.h` header file in the file that defines a UDF:

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
```

Alternatively, you can use the following line to include the `sqltypes_td.h` header file:

```
#include <sqltypes_td.h>
```

Using angle brackets (< >) or double quotation marks (" ") affects the path that the C preprocessor uses to search for the `sqltypes_td.h` header file.

## C/C++ Function Name

The name of the function in the C or C++ source code follows the C function naming conventions, with the additional restriction that the name cannot be longer than 30 characters.

In C++, the function definition must be preceded by `extern "C"` so that the function name is not converted to a C++ overloaded name. For example:

```
extern "C"
void f1( INTEGER *a, INTEGER *result, char sqlstate[6])
{
    ...
}
```

The function can call any module written in C++.

When you use the `CREATE FUNCTION` or `REPLACE FUNCTION` statement to install the UDF, you specify the name of the C or C++ function. For more information, see [Specifying the C/C++ Function Name](#).

On a Linux system, long names can sometimes cause errors when you install a UDF. For more information, see [Troubleshooting "Arg list too long" and "Argument list too long" Errors](#).

For information on how the name of the C or C++ function relates to UDF name overloading, see [Function Name Overloading](#).

## Function Parameters

### Parameter List

The list of parameters for a UDF is very specific. It includes the input parameters that Vantage passes when the UDF appears in a DML statement. It also includes output parameters that return the result of the UDF and the `SQLSTATE` result code.

A UDF can have 0 to 128 input parameters. The types that you use as input parameters and return type of a UDF correspond to SQL data types in the function call. For details, see [SQL Data Type Mapping](#).

Additional parameters might be required, depending on the parameter passing convention you choose.

Parameters are almost always specified as pointers to the data.

## Compatible Types

The argument types passed in to a UDF when it appears in a DML statement do not always have to be an exact match with the corresponding parameter declarations in the function definition, but they must be *compatible* and follow the precedence rules that apply to compatible types.

For example, you can define a function that declares an INTEGER parameter and you can successfully call the function with a BYTEINT argument because BYTEINT and INTEGER are compatible types and BYTEINT can fit into an INTEGER.

You cannot successfully call the function with a FLOAT argument, however, even though FLOAT is compatible with INTEGER, because of the precedence rules that apply to compatible types. A FLOAT cannot fit into an INTEGER without a possible loss of information.

Data types within the following groups are compatible, and appear in the order of precedence. A data type has precedence over the other data types that follow it in the list of compatible types.

Group	Compatible Type Precedence List
Character	<ul style="list-style-type: none"> <li>CHAR</li> <li>VARCHAR/CHAR VARYING/LONG VARCHAR</li> <li>CLOB</li> </ul>
Graphic	<ul style="list-style-type: none"> <li>CHARACTER CHARACTER SET GRAPHIC</li> <li>VARCHAR CHARACTER SET GRAPHIC or LONG VARCHAR CHARACTER SET GRAPHIC</li> </ul>
Byte	<ul style="list-style-type: none"> <li>BYTE</li> <li>VARBYTE</li> <li>BLOB</li> </ul>
Numeric	<ul style="list-style-type: none"> <li>BYTEINT</li> <li>SMALLINT</li> <li>INTEGER</li> <li>BIGINT</li> <li>DECIMAL/NUMERIC</li> <li>NUMBER</li> <li>REAL/FLOAT/DOUBLE PRECISION</li> </ul>

Data types separated by a slash ( / ) are synonyms and have the same precedence.

To define a function that accepts any numeric data type when you are not concerned with retaining exact precision, define the function parameter as REAL, FLOAT, or DOUBLE PRECISION. That way the function can accept any numeric data type because all numeric types are compatible and will be converted to the REAL data type before the function is called.

Data types that do not appear in the preceding table, such as DATE, TIME, and UDTs, are not compatible with any other data types.

To pass an argument that is not compatible with the corresponding parameter type, the function call must explicitly convert the argument to the proper type.

To provide a UDF that accepts argument types from different compatibility groups, you can do one of the following:

- Use TD\_ANYTYPE parameters, which can accept any system-defined data type or user-defined type (UDT). For more information, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).
- Use function name overloading, which allows you to define more than one function that has the same name, but declares different parameter data types. For more information, see [Function Name Overloading](#).

## Parameter Passing Convention

UDFs support two types of parameter passing conventions.

Parameter Passing Convention	Description	Usage
Parameter Style SQL	Provides a way to pass nulls as input arguments and return a null as the result.	<ul style="list-style-type: none"> <li>• Scalar function</li> <li>• Aggregate function</li> <li>• Table function</li> </ul>
Parameter Style TD_GENERAL	Does not accept null input arguments and does not return a null result.	<ul style="list-style-type: none"> <li>• Scalar function</li> <li>• Aggregate function</li> </ul>

The parameter passing convention you use to code a UDF must correspond to the parameter passing specification in the CREATE FUNCTION statement for the UDF.

IF CREATE FUNCTION specifies ...	AND you are writing ...	THEN use this syntax for function parameters ...
PARAMETER STYLE TD_GENERAL	a scalar function	<a href="#">Scalar Function Parameter Style TD_GENERAL Syntax</a>
	an aggregate function	<a href="#">Aggregate Function Parameter Style TD_GENERAL Syntax</a>
PARAMETER STYLE SQL or omits the PARAMETER STYLE option	a scalar function	<a href="#">Scalar Function Parameter Style SQL Syntax</a>
	an aggregate function	<a href="#">Aggregate Function Parameter Style SQL Syntax</a>
	a table function	<a href="#">Table Function Syntax</a>

Teradata provides a special UDF input parameter data type called the dynamic user-defined type (UDT). UDFs can specify up to eight dynamic UDT input parameters. A dynamic UDT can accommodate up to 128 attributes, where the data types of the attributes are determined at runtime.

If you consider each attribute of a dynamic UDT as an input parameter, then the number of effective input parameters increases to 1144, where:

$$(8 \text{ dynamic UDT parameters} * 128 \text{ attributes}) + (120 \text{ parameters of other data types}) = 1144$$

For details on using dynamic UDTs, see [Defining Functions that Use UDT Types](#).

## Function Name Overloading

UDFs support function name overloading: you can define several functions that have the same name but are different from each other in such a way to make each function unique.

The name that is overloaded is specified in the CREATE FUNCTION statement as the name to use to invoke the UDF from an SQL statement. You must provide a unique C or C++ function for each UDF with the same name.

You can also overload functions of different classes (scalar, aggregate, and table). That is, a scalar function can have the same name as an aggregate function within the same database as long as the two functions are unique.

## Characteristics of a Unique Function

Functions that have the same name are unique if any of the following is true:

- The number of parameters is different.
- The parameter data types are distinct from each other.

For the rules on which parameter data types are distinct from each other, see [Parameter Types in Overloaded Functions](#).

## Relationship to CREATE FUNCTION Statement

The name that is overloaded is the name that immediately follows the CREATE FUNCTION keywords in the CREATE FUNCTION statement.

Each time you use CREATE FUNCTION to overload a function name, you must specify:

- A parameter list that satisfies the characteristics of a unique function
- A unique name in the SPECIFIC clause
- A different C or C++ function

## Calling a Function That is Overloaded

For information on how Vantage determines which function to invoke when a user calls a function that is overloaded, see [Calling a Function That is Overloaded](#).

## Example: Overloaded Function with Distinct Parameter Data Types

Consider the following C functions I\_Sales and D\_Sales:

```
void I_Sales( INTEGER *quantity,
             INTEGER *result,
             char  sqlstate[6] )
{
    ...
}

void D_Sales( DECIMAL8 *quantity,
             INTEGER *result,
             char  sqlstate[6] )
{
    ...
}
```

The name you use in the CREATE FUNCTION statement for I\_Sales can be the same name you use in the CREATE FUNCTION statement for D\_Sales, because the data type of the **quantity** parameter for I\_Sales is distinct from the data type of the **quantity** parameter for D\_Sales.

Here are two SQL function definitions that overload the function name Sales:

```
CREATE FUNCTION Sales(Quantity DECIMAL(15,6))
RETURNS INTEGER
SPECIFIC D_Sales
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL;

CREATE FUNCTION Sales(Quantity INTEGER)
RETURNS INTEGER
SPECIFIC I_Sales
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL;
```

## Example: Overloaded Function with Different Number of Parameters

You can create the following two overloaded scalar and table functions successfully.

```
CREATE FUNCTION tbf (parameter_1 int)
  RETURNS int
  LANGUAGE C
  NO SQL
  SPECIFIC tbf1
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!tbf1!tbf1.c';
CREATE FUNCTION tbf(n INTEGER,n1 INTEGER,n2 INTEGER)
  RETURNS TABLE (c1 INTEGER, c2 INTEGER, c3 INTEGER)
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!tbf!tbf.c';
```

## Passing Dynamic UDTs as an Alternative to Overloading Function Names

Overloading a function provides applications with the ability to call one SQL UDF and pass arguments with data types that are determined at execution time.

Sometimes, however, defining C or C++ functions to handle many different combinations of parameter types can become cumbersome.

As an alternative to function overloading, consider defining a single C or C++ function with up to eight parameters that are defined as dynamic user-defined types (UDTs).

For details on using dynamic UDTs, see [Defining Functions that Use UDT Types](#).

## Using TD\_ANYTYPE Parameters as an Alternative to Overloading Function Names

You can use TD\_ANYTYPE parameters to reduce the number of overloaded functions needed to support routines that have constant parameter counts but differing parameter data types. TD\_ANYTYPE parameters can accept any system-defined data type or user-defined type (UDT); therefore, you can create a single function that can support a multitude of data types.

Consider the following function, which returns the ASCII numeric value of the first character of the input string:

```
CREATE FUNCTION ascii ( string_expr TD_ANYTYPE )
  RETURNS TD_ANYTYPE
  LANGUAGE C
  NO SQL
  SPECIFIC ascii
  EXTERNAL NAME 'CS!ascii!ascii.c'
  PARAMETER STYLE SQL;
```

The UDF writer can write this single function so that it accepts the following as valid argument types:

- CHARACTER
- VARCHAR
- CLOB
- UDT with a CHARACTER, VARCHAR, or CLOB attribute
- ARRAY with an element type of CHARACTER or VARCHAR

Similarly, the UDF writer can write the function so that the return type is BYTEINT, SMALLINT, or INTEGER. The function accepts the character set associated with the input string. If no character set is explicitly specified, the default character set is used.

A UDF writer can use TD\_ANYTYPE to represent a character data type of any chosen character set, or a decimal data type or interval type with any desired precision. By defining this function, you no longer have to create separate routines to perform the same basic function, such as `ascii_char_latin()`, `ascii_char_unicode()`, `ascii_varchar_latin()`, `ascii_varchar_unicode()`, `ascii_clob_latin()`, `ascii_clob_unicode()`, and so forth.

For more information about TD\_ANYTYPE parameters, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).

For the corresponding C code example for this function, see [C Scalar Function Using TD\\_ANYTYPE Parameters](#).

## Parameter Types in Overloaded Functions

If you overload function names, and the functions pass the same number of parameters, you must define the functions so that the parameter data types are distinct from each other. Use the following table to help you identify data types that are distinct from each other.

Types	Rules
Numeric	<p>The following numeric types are distinct from each other, and from other data types:</p> <ul style="list-style-type: none"> <li>• BYTEINT</li> <li>• SMALLINT</li> <li>• INTEGER</li> <li>• BIGINT</li> <li>• DECIMAL/NUMERIC</li> <li>• NUMBER</li> </ul>



Types	Rules
	<ul style="list-style-type: none"> <li>• REAL/FLOAT/DOUBLE PRECISION</li> </ul> DECIMAL, NUMERIC, or NUMBER types that differ in size are not distinct. For example, DECIMAL(6,2) and DECIMAL(8,3) are not distinct.
DateTime	The following DateTime types are distinct from each other, and from other data types: <ul style="list-style-type: none"> <li>• DATE</li> <li>• TIME</li> <li>• TIMESTAMP</li> <li>• TIME WITH TIME ZONE</li> <li>• TIMESTAMP WITH TIME ZONE</li> <li>• All INTERVAL types</li> </ul> TIME, TIMESTAMP, and INTERVAL types that differ in precision or fractional seconds precision are not distinct. For example, TIME(2) and TIME(6) are not distinct.
Character	The following character types are distinct from each other, and from other data types: <ul style="list-style-type: none"> <li>• VARCHAR/CHAR VARYING/LONG VARCHAR</li> <li>• CHAR</li> <li>• CLOB</li> </ul> Character strings that differ in length are not distinct. For example, CHAR(10) and CHAR(5) are not distinct. Character strings that specify different server character sets in the CHARACTER SET phrase are not distinct.
Byte	The following byte types are distinct from each other, and from other data types: <ul style="list-style-type: none"> <li>• BYTE</li> <li>• VARBYTE</li> <li>• BLOB</li> </ul>
Graphic	The following graphic types are distinct from each other, and from other data types: <ul style="list-style-type: none"> <li>• CHARACTER CHARACTER SET GRAPHIC</li> <li>• VARCHAR CHARACTER SET GRAPHIC or LONG VARCHAR CHARACTER SET GRAPHIC</li> </ul>
UDT	UDTs are distinct from other data types, including other UDTs.
Period	The following period types are distinct from each other, and from other data types: <ul style="list-style-type: none"> <li>• PERIOD(DATE)</li> <li>• PERIOD(TIME(<i>n</i>))</li> <li>• PERIOD(TIME(<i>n</i>) WITH TIME ZONE)</li> <li>• PERIOD(TIMESTAMP(<i>n</i>))</li> <li>• PERIOD(TIMESTAMP(<i>n</i>) WITH TIME ZONE)</li> </ul> PERIOD(TIME( <i>n</i> )), PERIOD(TIME( <i>n</i> ) WITH TIME ZONE), PERIOD(TIMESTAMP( <i>n</i> )), and PERIOD(TIMESTAMP( <i>n</i> ) WITH TIME ZONE) types that differ in fractional seconds precision are not distinct. For example, PERIOD(TIME(2)) and PERIOD(TIME(6)) are not distinct.

**Note:**

The types that are separated by a slash ( / ), such as DECIMAL/NUMERIC, are synonyms and are not distinct from each other.

## Scalar Function Parameter List

The parameter list must correspond to the parameter style in the CREATE FUNCTION statement for the UDF.

Use parameter style SQL to allow a UDF to pass nulls for the result or for input arguments; otherwise, use parameter style TD\_GENERAL.

When you use parameter style SQL, you must define indicator parameters for the result and for each parameter.

## Scalar Function Parameter Style TD\_GENERAL Syntax

```
void function_name (
    [ input_parameter_specification [...] ]
    result_type *result,
    char sqlstate[6]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

## Scalar Function Parameter Style SQL Syntax

```
void function_name (
    [ input_parameter_specification [...] ]
    result_type *result,
    [ indicator_parameter_specification [...] ]
    int indicator_result,
    char sqlstate[6]
    SQLTEXT function_name [m]
    SQLTEXT specific_function_name [l]
    SQLTEXT error_message [p]
)
```

```
{
    ...
}
```

***input\_parameter\_specification***

```
type *input_parameter,
```

***indicator\_parameter\_specification***

```
int *indicator_parameter,
```

## Scalar Function Parameter Syntax Elements

***function\_name***

Pointer to a C string whose value is the function name in the CREATE FUNCTION definition.

The function can use this name to build error messages.

***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE FUNCTION definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is the C type in `sqltypes_td.h` that corresponds to the SQL data type of *input\_parameter*.

***result***

Pointer to a data area big enough to hold the result that the function returns, as defined by the RETURNS clause in the corresponding CREATE FUNCTION statement.

***indicator\_parameter\_specification***

[Optional] Indicator parameter corresponding to an input parameter.

Each *input\_parameter\_specification* must have a corresponding *indicator\_parameter\_specification*. The input parameters and indicator parameters must be in the same order.

If the value of *indicator\_parameter* is -1, the value of the corresponding *input\_parameter* is null.

If the value of *indicator\_parameter* is 0, the value of the corresponding *input\_parameter* is a non-null value.

***indicator\_result***

Result indicator parameter corresponding to the result.

***sqlstate***

Pointer to a six-character C string that indicates the SQLSTATE value—success, exception, or warning. The first five characters are ASCII and the sixth is the C null character. The string is initialized to '00000', which indicates success.

For more information on SQLSTATE values, see [Returning SQLSTATE Values](#).

***m***

Number of characters in the function name in the CREATE FUNCTION definition. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***specific\_function\_name***

Pointer to a C string whose value is the name of the external function being invoked.

If the CREATE FUNCTION statement includes the SPECIFIC clause, *specific\_function\_name* is the name in the SPECIFIC clause; otherwise, *specific\_function\_name* is the same as *function\_name*.

The function can use this name to build error messages.

***l***

Number of characters in the name of the external function. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***error\_message***

Pointer to a C string whose value is the the error message text.

***p***

Number of characters in the error message text. The maximum value is 256.

**Related Information:**

[Argument and Result Behavior](#)

**Example: Scalar Function with Parameter Style TD\_GENERAL**

Here is a code excerpt that shows how to declare a scalar function that uses parameter style TD\_GENERAL:

```

/***** C source file name: substr.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void udf_scalar_substr( VARCHAR_LATIN *inputString,
                        INTEGER          *start,
                        VARCHAR_LATIN *result,
                        char             sqlstate[6])
{
    ...
}

```

For a complete example of the scalar function, see [Example: Basic Scalar Function](#).

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION udfSubStr
  (inputString VARCHAR(512),
   start        INTEGER)
RETURNS VARCHAR(512)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!substr!udfsrc/substr.c!F!udf_scalar_substr'
PARAMETER STYLE TD_GENERAL;

```

## Example: Scalar Function with Parameter Style SQL

Here is an example of how to declare a scalar function that uses parameter style SQL:

```

/***** C source file name: substr.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void udf_scalar_substr( VARCHAR_LATIN *inputString,
                        INTEGER          *start,
                        VARCHAR_LATIN *result,
                        int              *inputStringIsNull,
                        int              *startIsNull,
                        int              *resultIsNull,

```

```

        char          sqlstate[6],
        SQL_TEXT      extname[129],
        SQL_TEXT      specific_name[129],
        SQL_TEXT      error_message[257] )
{
    ...
}

```

For a complete example of the scalar function, see [Example: Basic Scalar Function](#).

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION udfSubStr
  (strex VARCHAR(512),
   n1 INTEGER)
RETURNS VARCHAR(512)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!substr!udfsrc/substr.c!F!udf_scalar_substr'
PARAMETER STYLE SQL;

```

## Scalar Function Body

### Basic Scalar Function Definition

Here are the basic steps you take to define a scalar function:

1. Define the SQL\_TEXT constant.  
For more information, see [SQL\\_TEXT Definition](#).
2. Include the sqltypes\_td.h header file.  
For more information, see [Header Files](#).
3. Include other header files that define macros and variables that the function uses.
4. Define the function parameter list in the order that matches the parameter passing convention specified in the CREATE FUNCTION statement.  
For more information, see [Scalar Function Parameter List](#).
5. Implement the function and set the result to the appropriate value.
6. If the function detects an error, set the:
  - *sqlstate* argument to an SQLSTATE exception or warning condition before the function exits.  
For more information, see [Returning SQLSTATE Values](#).

- *error\_message* string to the error message text. The characters must be inside the LATIN character range. The string is initialized to a null-terminated string on input.

7. If the function uses parameter style SQL, set the *indicator\_result* argument.

IF the result is ...	THEN set the <i>indicator_result</i> argument to ...
NULL	-1.
not NULL	0.

## Example: Basic Scalar Function

Here is an example of a scalar function that extracts a substring from an input string given the starting position of the substring. The scalar function uses parameter style SQL.

```

/***** C source file name: substr.c *****/
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#define IsNull      -1
#define IsNotNull   0
void udf_scalar_substr( VARCHAR_LATIN *inputString,
                        INTEGER        *start,
                        VARCHAR_LATIN *result,
                        int             *inputStringIsNull,
                        int             *startIsNull,
                        int             *resultIsNull,
                        char            sqlstate[6],
                        SQL_TEXT        extname[129],
                        SQL_TEXT        specific_name[129],
                        SQL_TEXT        error_message[257] )
{
    int inputStrLength;
    /* Return Null value on Null Input */
    if ((*inputStringIsNull == IsNull))
    {
        strcpy(sqlstate, "22004");
        strcpy((char *) error_message, "Null value not allowed.");
        *resultIsNull = IsNull;
        return;
    }
    strcpy((char *) error_message, " ");
    *resultIsNull = IsNotNull;
    inputStrLength = strlen((const char *) inputString);

```

```

if (*start < 1)
{
    strcpy((char *) result, (const char *) inputString) ;
    return;
}
if (*start > inputStrLength)
{
    strcpy((char *) result, "");
    return;
}
strcpy((char *) result,
        (const char *) inputString + (*start - 1));
}

```

## Aggregate Function Parameter List

The parameter list must correspond to the parameter style in the CREATE FUNCTION statement for the UDF.

Use parameter style SQL to allow a UDF to pass null values for the result or for input arguments; otherwise, use parameter style TD\_GENERAL.

When you use parameter style SQL, you must define indicator parameters for the result and for each parameter.

## Aggregate Function Parameter Style TD\_GENERAL Syntax

```

void  function_name (
    FNC_Phase  aggregation_phase,
    FNC_Context_t  *function_context,
    [ input_parameter_specification [...] ],
    result_type  *result,
    char  sqlstate[6]
)
{
    ...
}

```

***input\_parameter\_specification***

```

type  *input_parameter,

```



## Aggregate Function Parameter Style SQL Syntax

```
void function_name (
    FNC_Phase aggregation_phase,
    FNC_Context_t *function_context,
    [ input_parameter_specification [...] ]
    result_type *result,
    [ indicator_parameter_specification [...] ]
    int indicator_result,
    char sqlstate[6]
    SQLTEXT function_name [m]
    SQLTEXT specific_function_name [l]
    SQLTEXT error_message [p]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

### *indicator\_parameter\_specification*

```
int *indicator_parameter,
```

## Aggregate Function Parameter Syntax Elements

### *function\_name*

Pointer to a C string whose value is the function name in the CREATE FUNCTION definition.

The function can use this name to build error messages.

### *aggregation\_phase*

Current aggregation phrase, which determines how the aggregate function processes the input data.

The aggregation phrase definition is:

```
typedef enum {
    AGR_INIT      = 1,
```

```

    AGR_DETAIL    = 2,
    AGR_COMBINE   = 3,
    AGR_FINAL     = 4,
    AGR_NODATA    = 5
} FNC_Phase;

```

<i>aggregation_phase</i> Value	How Function Processes Input Data
AGR_INIT	Function allocates and initializes intermediate storage for accumulating summary information.
AGR_DETAIL	Function accumulates input data into intermediate storage area for specified group.
AGR_COMBINE	Function combines intermediate storage areas for each group being aggregated.
AGR_FINAL	Function produces final aggregate result for group.
AGR_NODATA	Function produces result for null aggregate set.

### ***function\_context***

Pointer to the function context structure, the intermediate aggregate storage area for groups on which the aggregate function operates.

The function context structure definition is:

```

typedef struct FNC_Context_t {
    int          version;
    FNC_flags_t  flags;
    void         *interim1;
    int          intrm1_length;
    void         *interim2;
    int          intrm2_length;
    long         group_count;
    long         window_size;
    long         pre_window;
    long         post_window;
} FNC_Context_t;

```

<b>FNC_Context_t Member</b>	<b>Description</b>
<i>version</i>	Version of context structure. Current version is 1.
<i>interim1</i>	Pointer to aggregate storage area that saves intermediate results.
<i>intrm1_length</i>	Length of <i>interim1</i> .

FNC_Context_t Member	Description
<i>interim2</i>	Pointer to intermediate storage area to combine with <i>interim1</i> storage area when value of <i>aggregation_phase</i> is AGR_COMBINE.
<i>intrm2_length</i>	Length of <i>interim2</i> .

FNC\_Context\_t members not defined above are reserved for future use.

### ***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE FUNCTION definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is the C type in `sqltypes_td.h` that corresponds to the SQL data type of *input\_parameter*.

### ***result***

Pointer to a data area big enough to hold the result that the function returns, as defined by the RETURNS clause in the corresponding CREATE FUNCTION statement.

### ***indicator\_parameter\_specification***

[Optional] Indicator parameter corresponding to an input parameter.

Each *input\_parameter\_specification* must have a corresponding *indicator\_parameter\_specification*. The input parameters and indicator parameters must be in the same order.

If the value of *indicator\_parameter* is -1, the value of the corresponding *input\_parameter* is null.

If the value of *indicator\_parameter* is 0, the value of the corresponding *input\_parameter* is a non-null value.

### ***indicator\_result***

Result indicator parameter corresponding to the result.

### ***sqlstate***

Pointer to a six-character C string that indicates the SQLSTATE value—success, exception, or warning. The first five characters are ASCII and the sixth is the C null character. The string is initialized to '00000', which indicates success.

For more information on SQLSTATE values, see [Returning SQLSTATE Values](#).

***m***

Number of characters in the function name in the CREATE FUNCTION definition. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***specific\_function\_name***

Pointer to a C string whose value is the name of the external function being invoked.

If the CREATE FUNCTION statement includes the SPECIFIC clause, *specific\_function\_name* is the name in the SPECIFIC clause; otherwise, *specific\_function\_name* is the same as *function\_name*.

The function can use this name to build error messages.

***l***

Number of characters in the name of the external function. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***error\_message***

Pointer to a C string whose value is the the error message text.

***p***

Number of characters in the error message text. The maximum value is 256.

**Related Information:**

[Argument and Result Behavior](#)

## Example: Aggregate Function with Parameter Style TD\_GENERAL

This aggregate function declaration uses parameter style TD\_GENERAL:

```

/***** C source file name: STD_DEV.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <math.h>
void STD_DEV ( FNC_Phase      phase,
               FNC_Context_t *fctx,
               FLOAT          *x,
               FLOAT          *result,
```

```

        char          sqlstate[6] )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION STD_DEV(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL;

```

## Example: Aggregate Function with Parameter Style SQL

This example declares an aggregate function that uses parameter style SQL:

```

/***** C source file name: STD_DEV.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <math.h>

void STD_DEV ( FNC_Phase      phase,
               FNC_Context_t *fctx,
               FLOAT          *x,
               FLOAT          *result,
               int             *x_i,
               int             *result_i,
               char            sqlstate[6],
               SQL_TEXT        *function_name,
               SQL_TEXT        *specific_name,
               SQL_TEXT        error_message[257])
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```
CREATE FUNCTION STD_DEV(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL;
```

## Aggregate Function Body

### Basic Algorithm of an Aggregate Function

Vantage invokes an aggregate UDF once for each item in an aggregation group, passing the detail values of a group through the input arguments. To accumulate summary information, an aggregate function must retain context each time it is called.

You do not need to understand or worry about how to create a group or how to code an aggregate UDF to deal with groups, because Teradata takes care of that. You only need to write the basic algorithm of combining the data passed to it to produce the desired result.

The following table describes the flow of an aggregate function for a group. To see the flow of a window aggregate function, see [Window Aggregate Function Body](#).

Phase	Value of <i>aggregation_phase</i> Argument	Description
1	AGR_INIT	This is the first time Vantage invokes the aggregate UDF for an aggregation group. The function must: <ul style="list-style-type: none"> <li>• Allocate intermediate storage and initialize it.</li> <li>• Process the first detail passed into the function.</li> </ul>
2	AGR_DETAIL	In this phase, Vantage calls the function once for each row to be aggregated for each group. The function must combine the function argument input data with the intermediate storage defined for the group.
3	AGR_COMBINE	This phase combines results from different AMPs for a specific group. The function must combine the data for the two intermediate storage areas passed in.
4	AGR_FINAL	No more input is expected for the group. The function must produce the final result for the group.
5	AGR_NODATA	This phase is only presented when there is absolutely no data to aggregate.

## Using a Switch Statement for the Basic Algorithm

One way to write an aggregate UDF to execute the proper code required for an aggregation phase is to use the C switch statement.

To help illustrate what is required, the discussion uses code excerpts from a simple aggregate UDF that calculates the standard deviation. For the complete code example, see [C Aggregate Function](#).

Here is an example:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <math.h>
void STD_DEV ( FNC_Phase      phase,
               FNC_Context_t *fctx,
               FLOAT          *x,
               FLOAT          *result,
               int             *x_i,
               int             *result_i,
               char            sqlstate[6],
               SQL_TEXT        fncname[129],
               SQL_TEXT        sfncname[129],
               SQL_TEXT        error_message[257])
{
    /* switch to determine the aggregation phase */
    switch (phase)
    {
        /* This phase is executed once per group and */
        /* allocates and initializes intermediate */
        /* aggregate storage */
        case AGR_INIT:
            /* Get storage for intermediate aggregate values. */
            ...
            /* Initialize the intermediate aggregate values */
            ...
            /******
            /* Fall through to detail phase, because the */
            /* AGR_INIT phase passes in the first set of */
            /* values for the group. */
            /******
            /* This phase is executed once for each selected */
            /* row to aggregate. One copy will run on each AMP. */
            case AGR_DETAIL:
```

```

    ...
    break;
/* This phase combines the results of ALL          */
/* individual AMPs for each group.                  */
case AGR_COMBINE:
    ...
    break;
/* This phase returns the final result. */
/* It is called once for each group.    */
case AGR_FINAL:
{
    ...
    break;
}
case AGR_NODATA:
    /* return null if no data */
    *result_i = -1;
    break;
default:
    /* If it gets here there must be an error because this */
    /* function does not accept any other phase options */
    strcpy(sqlstate, "U0005");
}
return;
}

```

## Related Information

FOR more information on ...	SEE ...
allocating and initializing intermediate storage in AGR_INIT phase	<a href="#">Allocating Storage</a> and <a href="#">Initializing Storage</a> .
accumulating intermediate results in the AGR_DETAIL phase	<a href="#">Saving Intermediate Results</a> .
combining intermediate storage areas from different AMPs	<a href="#">Combining Intermediate Storage Areas</a> .
returning the final group result	<a href="#">Producing the Final Group Result</a> .

## Window Aggregate Function Parameter List

The parameter list must correspond to the parameter style in the CREATE FUNCTION statement for the UDF.



Use parameter style SQL to allow a UDF to pass nulls for the result or for input arguments; otherwise, use parameter style TD\_GENERAL.

When you use parameter style SQL, you must define indicator parameters for the result and for each parameter.

## Window Aggregate Function Parameter Style TD\_GENERAL Syntax

```
void function_name (
    FNC_Phase aggregation_phase,
    FNC_Context_t *function_context,
    [ input_parameter_specification [...] ],
    result_type *result,
    char sqlstate[6]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

## Window Aggregate Function Parameter Style SQL Syntax

```
void function_name (
    FNC_Phase aggregation_phase,
    FNC_Context_t *function_context,
    [ input_parameter_specification [...] ]
    result_type *result,
    [ indicator_parameter_specification [...] ]
    int indicator_result,
    char sqlstate[6]
    SQLTEXT function_name [m]
    SQLTEXT specific_function_name [l]
    SQLTEXT error_message [p]
)
{
    ...
}
```

***input\_parameter\_specification***

```
type *input_parameter,
```

***indicator\_parameter\_specification***

```
int *indicator_parameter,
```

## Window Function Parameter Syntax Elements

***function\_name***

Pointer to a C string whose value is the function name in the CREATE FUNCTION definition.

The function can use this name to build error messages.

***aggregation\_phase***

Current aggregation phrase, which determines how the aggregate function processes the input data.

The aggregation phrase definition is:

```
typedef enum {
    AGR_INIT          = 1,
    AGR_DETAIL        = 2,
    AGR_COMBINE       = 3,
    AGR_FINAL         = 4,
    AGR_NODATA        = 5,
    AGR_MOVINGTRAIL   = 6
} FNC_Phase;
```

For details about the aggregation phases, see [Basic Algorithm of a Window Aggregate Function](#).

***function\_context***

Pointer to the function context structure, the intermediate aggregate storage area for groups on which the aggregate function operates.

The function context structure definition is:

```
typedef struct FNC_Context_t {
    int          version;
    FNC_flags_t  flags;
    void         *interim1;
```

```

int      intrm1_length;
void     *interim2;
int      intrm2_length;
long     group_count;
long     window_size;
long     pre_window;
long     post_window;
} FNC_Context_t;

```

Vantage sets up the following fields before invoking the function:

- window\_size
- pre\_window
- post\_window

The function uses the fields to define a window of rows for the function to operate on.

For details about the FNC\_Context\_t fields that are applicable to window aggregate functions, see [FNC\\_Context\\_t Fields](#).

### ***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE FUNCTION definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is the C type in `sqltypes_td.h` that corresponds to the SQL data type of *input\_parameter*.

### ***result***

Pointer to a data area big enough to hold the result that the function returns, as defined by the RETURNS clause in the corresponding CREATE FUNCTION statement.

### ***indicator\_parameter\_specification***

[Optional] Indicator parameter corresponding to an input parameter.

Each *input\_parameter\_specification* must have a corresponding *indicator\_parameter\_specification*. The input parameters and indicator parameters must be in the same order.

If the value of *indicator\_parameter* is -1, the value of the corresponding *input\_parameter* is null.

If the value of *indicator\_parameter* is 0, the value of the corresponding *input\_parameter* is a non-null value.

***indicator\_result***

Result indicator parameter corresponding to the result.

***sqlstate***

Pointer to a six-character C string that indicates the SQLSTATE value—success, exception, or warning. The first five characters are ASCII and the sixth is the C null character. The string is initialized to '00000', which indicates success.

For more information on SQLSTATE values, see [Returning SQLSTATE Values](#).

***m***

Number of characters in the function name in the CREATE FUNCTION definition. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***specific\_function\_name***

Pointer to a C string whose value is the name of the external function being invoked.

If the CREATE FUNCTION statement includes the SPECIFIC clause, *specific\_function\_name* is the name in the SPECIFIC clause; otherwise, *specific\_function\_name* is the same as *function\_name*.

The function can use this name to build error messages.

***l***

Number of characters in the name of the external function. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***error\_message***

Pointer to a C string whose value is the the error message text.

***p***

Number of characters in the error message text. The maximum value is 256.

#### Related Information:

[Argument and Result Behavior](#)

## FNC\_Context\_t Fields

The following table identifies the FNC\_Context\_t fields that are used by window aggregate functions. If a field is not listed in the table, it is not applicable for window aggregate functions or the field is reserved for future use.

Fields	Specifies
version	the version of the context structure.
interim1	a pointer to the aggregate storage area that saves intermediate results.
intrm1_length	the length of interim1.
window_size	<ul style="list-style-type: none"> <li>For the cumulative window type, this value is -1.</li> <li>For the reporting window type, the value is -2.</li> <li>For the moving window type, the value is set as <i>post_window - pre_window + 1</i> (+1 for the current row).</li> </ul>
pre_window	<p>This value is specified as part of the PRECEDING clause. The value of this field is negative if this points to a row that precedes the current row.</p> <p>This field is not applicable for the cumulative and reporting window types. It is initialized to zero in those cases.</p>
post_window	<p>This value is specified as part of the FOLLOWING clause. The value of this field is positive if this points to a row that follows the current row.</p> <p>This field is not applicable for the cumulative and reporting window types. It is initialized to zero in those cases.</p>

Vantage sets up the following fields prior to invoking the function. The fields are used to define a window of rows for the function to operate on.

- window\_size
- pre\_window
- post\_window

## Example: Window Aggregate Function with Parameter Style TD\_GENERAL

This aggregate function declaration uses parameter style TD\_GENERAL:

```

/***** C source file name: STD_DEV.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <math.h>
void STD_DEV ( FNC_Phase      phase,
               FNC_Context_t *fctx,
               FLOAT          *x,
               FLOAT          *result,
               char           sqlstate[6] )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION STD_DEV(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL;

```

## Example: Window Aggregate Function with Parameter Style SQL

This example declares an aggregate function that uses parameter style SQL:

```

/***** C source file name: STD_DEV.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <math.h>

void STD_DEV ( FNC_Phase      phase,
               FNC_Context_t *fctx,
               FLOAT          *x,
               FLOAT          *result,
               int            *x_i,
               int            *result_i,
               char           sqlstate[6],

```

```

        SQL_TEXT      *function_name,
        SQL_TEXT      *specific_name,
        SQL_TEXT      error_message[257])
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION STD_DEV(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL;

```

## Window Aggregate Function Body

### Supported Window Types

You can apply a window specification to an aggregate function. The following window types are supported for user-defined aggregate functions.

Window Type	Aggregation Group	Supported Partitioning Strategy
Reporting window	ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	Hash partitioning
Cumulative window	<ul style="list-style-type: none"> <li>ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW</li> <li>ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING</li> </ul>	Hash partitioning
Moving window	<ul style="list-style-type: none"> <li>ROWS BETWEEN <i>value</i> PRECEDING AND CURRENT ROW</li> <li>ROWS BETWEEN CURRENT ROW AND <i>value</i> FOLLOWING</li> <li>ROWS BETWEEN <i>value</i> PRECEDING AND <i>value</i> FOLLOWING</li> <li>ROWS BETWEEN <i>value</i> PRECEDING AND <i>value</i> PRECEDING</li> <li>ROWS BETWEEN <i>value</i> FOLLOWING AND <i>value</i> FOLLOWING</li> </ul>	Hash partitioning and range partitioning

## Unsupported Window Types

The following window types are *not* supported for user-defined aggregate functions.

Window Type	Aggregation Group
Moving window	<ul style="list-style-type: none"> <li>ROWS BETWEEN UNBOUNDED PRECEDING AND <i>value</i> FOLLOWING</li> <li>ROWS BETWEEN <i>value</i> PRECEDING AND UNBOUNDED FOLLOWING</li> </ul>

## Partitioning

The range partitioning strategy helps to avoid hot AMP situations where the values of the columns of the PARTITION BY clause result in the distribution of too many rows to the same partition or AMP.

Range and hash partitioning is supported for moving window types. Only hash partitioning is supported for the reporting and cumulative window types because of potential ambiguities that can occur when a user tries to reference previous values assuming a specific ordering within window types like reporting and cumulative, which are semantically not order dependent.

You should make sure that the function uses an appropriate set of column values for the PARTITION BY clause to avoid potential skew situations for the reporting or cumulative aggregate cases. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Basic Algorithm of a Window Aggregate Function

The following table describes the flow of a window aggregate function for a group.

Phase	Value of <i>aggregation_phase</i> Argument	Description
1	AGR_INIT	<p>This phase is triggered once per partition when starting evaluation of a new aggregation group.</p> <p>The function must:</p> <ul style="list-style-type: none"> <li>Allocate and initialize intermediate storage for accumulating data passed in during the various aggregation phases.</li> <li>Save the first set of data values passed in through arguments into the intermediate storage area.</li> </ul>
2	AGR_DETAIL	<p>This phase is triggered every time the forward row progresses.</p> <p>Vantage calls the function once for each row to be aggregated for each group.</p> <p>The function must accumulate the argument input data into the intermediate storage defined for the specific group.</p>
3	AGR_COMBINE	<p>This phase is not applicable for window aggregate functions, but may be included to generate an error if this phase is reached erroneously.</p>



Phase	Value of <i>aggregation_phase</i> Argument	Description
		This phase is used for aggregate UDFs. For more information, see <a href="#">Aggregate Function Parameter List</a> .
6	AGR_MOVINGTRAIL	<p>This phase is applicable for the moving window type.</p> <p>This phase is triggered only by the last few rows of a moving window when the forward pointer to the window reaches the end of the partition.</p> <p>The phase does not provide any new values to the function, but indicates to the function that processing has reached the end of the partition.</p> <p>The function can use this phase to adjust the necessary internal count and offset values to reflect the actual size as the window diminishes towards the end of the partition.</p>
4	AGR_FINAL	<p>This phase is invoked at the time the final evaluated result needs to be moved into the result row.</p> <p>No more input is expected for the group, and the function produces the final aggregate result for the group.</p>
5	AGR_NODATA	<p>This phase is only presented when there is absolutely no data to aggregate.</p> <p>The function provides a result for a null aggregate set.</p>

You must maintain a cache of rows corresponding to the window size. It may be useful to maintain a counter value that indicates the total rows read so far. For example, if C represents the counter, then the window of rows for evaluation would be:

```
if (C < window_size),
    window size would be C
else
    window size would be as indicated in function context.
```

The base of the window would be C - window\_size and the end of the window would correspond to C.

The complexities of the various window combinations (PRECEDING/FOLLOWING/CURRENT, etc) are handled by Vantage. You only need to maintain the window cache of rows and implement the semantics of the function on this cache of rows.

## Usage Notes

When developing a window aggregate function, be aware of the following:

- When specifying the return type of the function, you should consider the allowable window sizes and select an appropriate return data type to avoid overflows.
- Vantage does not validate if a window aggregate UDF is used in a specific window type, so you may want to include window type checking code into your function.

## Intermediate Aggregate Storage

All aggregate and window aggregate UDFs require intermediate storage to combine data passed in during the various aggregation phases.

This topic describes how an aggregate UDF handles intermediate storage during each aggregation phase. To help illustrate what is required, the discussion uses code excerpts from a simple aggregate UDF that calculates the standard deviation. For the complete code example, see [C Aggregate Function](#).

## Function Context Parameter

As discussed in [Aggregate Function Parameter List](#), the aggregate UDF parameter list includes a pointer to an `FNC_Context_t` function context structure, which is defined as:

```
struct FNC_Context_t {
    int          version;
    FNC_flags_t  flags;
    void         *interim1;
    int          intrm1_length;
    void         *interim2;
    int          intrm2_length;
    long         group_count;
    long         window_size;
    long         pre_window;
    long         post_window;
} FNC_Context_t;
```

The function should treat the structure as read only, and must never change any of its values. The structure has four members that are used for intermediate storage in aggregate functions.

Member ...	Specifies ...
<i>interim1</i>	a pointer to the aggregate storage area in which the function saves intermediate results.
<i>interim2</i>	a pointer to an intermediate storage area to combine with the <i>interim1</i> storage area when the value of aggregation phase is AGR_COMBINE.
<i>intrm1_length</i>	the length of <i>interim1</i> .
<i>intrm2_length</i>	the length of <i>interim2</i> .

## Using a C Structure to Access the Storage Area

The type of intermediate results that an aggregate function needs to save depends on the type of calculation that the function performs.

Consider a standard deviation function STD\_DEV(x) that uses the following equation:

$$s = \sqrt{\frac{\sum X^2}{N} - \left(\frac{\sum X}{N}\right)^2}$$

Based on the calculation, the function needs to store the following:

- N
- sum( $X^2$ )
- sum(X)

The function can declare a C structure with components that match the necessary intermediate values:

```
typedef struct agr_storage {
    FLOAT n;
    FLOAT x_sq;
    FLOAT x_sum;
} AGR_Storage;
```

The function can then define a pointer to AGR\_Storage that points to the FNC\_Context\_t.interim1 function argument.

Suppose the function declaration uses the following parameters:

```
void STD_DEV ( FNC_Phase    phase,
               FNC_Context_t *fctx,
               FLOAT        *x,
               FLOAT        *result,
               char          sqlstate[6] )
{
    ...
}
```

The following statement defines a pointer to AGR\_Storage named s1 that points to the FNC\_Context\_t.interim1 function argument:

```
AGR_Storage *s1 = fctx->interim1;
```

The function can then easily access the storage area using the pointer to AGR\_Storage. For example:

```
s1->n      = 0;
s1->x_sq   = 0;
s1->x_sum  = 0;
```

## Allocating Storage

During the AGR\_INIT aggregation phase, a UDF must allocate the intermediate storage pointed to by `FNC_Context_t.interim1`.

To allocate the required memory, the UDF calls the `FNC_DefMem` library function, specifying the size of the structure, which has a 64000 byte maximum. The UDF cannot request more memory than the value of *interim\_size* in the CLASS AGGREGATE clause of the CREATE FUNCTION statement. If *interim\_size* is omitted, then the limit is 64 bytes.

For best performance, allocate only enough memory to satisfy the needs of the function for intermediate storage.

In the standard deviation function example, the size of the structure can be computed by using `sizeof(AGR_Storage)`. The following statement allocates the intermediate storage pointed to by `FNC_Context_t.interim1`:

```
s1 = FNC_DefMem(sizeof(AGR_Storage));
```

`FNC_DefMem` returns a NULL pointer for any of the following conditions:

- The aggregate function asks for more memory than the value of *interim\_size* in the CLASS AGGREGATE clause of the CREATE FUNCTION statement
- The CLASS AGGREGATE clause of the CREATE FUNCTION statement does not specify a value for *interim\_size* and the aggregate function asks for more than the default 64 bytes
- The function asks for more than the 64000 byte maximum

If the UDF cannot complete without the memory, then the function can return an error by setting up the SQLSTATE error results argument.

## Initializing Storage

During the AGR\_INIT aggregation phase, a UDF must initialize the intermediate storage pointed at by the returned pointer from the `FNC_DefMem` call.

For example, the following statements initialize the intermediate storage area in the standard deviation example:

```
s1->n      = 0;
s1->x_sq   = 0;
s1->x_sum  = 0;
```

## Saving Intermediate Results

During the AGR\_INIT aggregation phase, a UDF must also save the first set of data values passed in through arguments into the intermediate aggregate storage area. The first set of data values are for the first row to aggregate for the group.

Thereafter, each time the function is called during the AGR\_DETAIL aggregation phase, it must accumulate the row data passed in through arguments into the intermediate aggregate storage area for the specific group. Each group being aggregated to has a separate intermediate storage area.

The aggregation storage area is pointed to by interim1 in the FNC\_Context\_t function context argument.

In the standard deviation example, the x input argument is the column the standard deviation is being calculated for. For the standard deviation calculation, the function uses the value of x to calculate the following:

- $\text{sum}(X^2)$
- $\text{sum}(X)$

The following statements perform the necessary calculations using the column value, and then combine the results with the intermediate storage:

```
s1->n++;
s1->x_sq += *x * *x;
s1->x_sum += *x;
```

## Combining Intermediate Storage Areas

During the AGR\_COMBINE aggregation phase, a UDF must combine two intermediate aggregate storage areas into one storage area for each group that is being aggregated. The storage areas that the function is combining are the results from different AMPs for a specific group.

The storage areas are pointed to by interim1 and interim2 in the FNC\_Context\_t function context structure parameter. The pointer in interim2 points to the aggregate storage area to combine with the aggregate storage area specified by interim1.

The ultimate result is to create one summary aggregate storage area for each group on which the aggregate function operates.

In the standard deviation example, the function can define another pointer to AGR\_Storage that points to the FNC\_Context\_t.interim2 function argument. If the argument name is fctx, the following statement defines a pointer to AGR\_Storage named s2:

```
AGR_Storage *s2 = fctx->interim2;
```

The following statements combine the aggregate storage areas:

```
s1->n      += s2->n;
s1->x_sq   += s2->x_sq;
s1->x_sum  += s2->x_sum;
```

**Note:**  
The AGR\_COMBINE aggregation phase is not applicable for window aggregate functions.

## Producing the Final Group Result

During the AGR\_FINAL aggregation phase, a UDF produces the final result for the group. The function must set the result argument to the final value.

The storage area pointed to by interim1 in the FNC\_Context\_t argument points to the aggregate storage area for the group. The function uses the interim1 pointer to calculate the final value and return it in the result argument.

In the standard deviation example, the function can use the following statements to calculate the final value:

```
FLOAT term2      = s1->x_sum / s1->n;
FLOAT variance = (s1->x_sq / s1->n) - (term2 * term2);
```

The following statement sets the result argument to the final value of the standard deviation:

```
*result = sqrt(variance);
```

## Aggregate Cache and Aggregate UDFs

To enhance performance, Vantage caches intermediate aggregate storage areas. The size of the aggregate cache is fixed at 1 MB.

If the number of intermediate storage areas exceeds the capacity of the aggregate cache, Vantage pages the least recently used intermediate storage areas to a spool file on disk, which causes a larger number of aggregation phases to occur and impacts performance.

For details on improving aggregate cache usage for an aggregate UDF, including how to determine the maximum number of intermediate storage areas, see the information about CREATE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Related Information

FOR more information on ...	SEE ...
the FNC_DefMem library function	<a href="#">FNC_DefMem</a> .

FOR more information on ...	SEE ...
example code that shows each aggregation phase	the UDF code example in <a href="#">C Aggregate Function</a> .
example code that shows an aggregate function using LOBs	the UDF code example in <a href="#">C Aggregate Function Using LOBs</a> .
example code that shows an aggregate function using UDTs	the UDF code example in <a href="#">C Aggregate Function Using UDTs</a> .
example code that shows a window aggregate function	the UDF code example in <a href="#">C Window Aggregate Function</a> .
invoking window aggregate UDFs	<i>Teradata Vantage™ - SQL Functions, Expressions, and Predicates</i> , B035-1145.
optimizing the aggregate cache	CREATE FUNCTION information in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.

## Table Function Parameter List

The parameter list of a table function is similar to the parameter list of a scalar function. The main difference is that the table function parameter list can include multiple result parameters because a table function returns a row result instead of just one value.

A table function can have:

- 128 input parameters
- as many result parameters as defined by the RETURNS TABLE clause of the CREATE FUNCTION or REPLACE FUNCTION statement for the table function.

The number of result parameters is limited by the maximum number of columns that can be defined for a regular table.

---

### Note:

The data type of a result parameter in a table function cannot be TD\_ANYTYPE.

---

## Fixed Result Row Specification

If the CREATE FUNCTION or REPLACE FUNCTION statement specifies a column list for the table function row result, then the table function is said to have a fixed result row specification. The number of result parameters in the table function parameter list matches the number of columns in the column list in the RETURNS TABLE clause of the CREATE FUNCTION or REPLACE FUNCTION statement.

Here is an example of a CREATE FUNCTION statement for a table function with a fixed result row specification:

```
CREATE FUNCTION getStoreData
  (FileToRead INTEGER)
RETURNS TABLE (StoreNo INTEGER, ItemNo INTEGER, QuantSold INTEGER)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getdata!udfsrc/g1.c'
PARAMETER STYLE SQL;
```

In this example, the number of result parameters in the table function parameter list is three.

## Dynamic Result Row Specification

If the CREATE FUNCTION or REPLACE FUNCTION statement specifies a RETURNS TABLE VARYING COLUMNS clause, then the table function is said to have a dynamic result row specification. The number of result parameters in the table function parameter list matches the maximum number of columns in the RETURNS TABLE VARYING COLUMNS clause of the CREATE FUNCTION or REPLACE FUNCTION statement.

The actual number and data types of the result parameters that the table function needs to return values in is specified dynamically at runtime in the SELECT statement that invokes the table function.

Here is an example of a CREATE FUNCTION statement for a table function with a dynamic result row specification:

```
CREATE FUNCTION getStoreData
  (FileToRead INTEGER)
RETURNS TABLE VARYING COLUMNS (10)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getdata!udfsrc/g2.c'
PARAMETER STYLE SQL;
```

In this example, the number of result parameters in the table function parameter list is 10.

## Parameter Style

The CREATE FUNCTION or REPLACE FUNCTION statement for a table UDF must specify parameter style SQL, which allows the table UDF to pass nulls for the result and for input arguments.

Parameter style SQL requires you to declare indicator parameters for each result and for each input parameter.



## Table Function Syntax

```
void function_name (
  [ input_parameter_specification [...] ]
  result_specification [...]
  [ indicator_parameter_specification [...] ]
  [ indicator_result_specification [...] ]
  char sqlstate[6]
  SQLTEXT function_name [m]
  SQLTEXT specific_function_name [l]
  SQLTEXT error_message [p]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

### *result\_specification*

```
type *result,
```

### *indicator\_parameter\_specification*

```
int *indicator_parameter,
```

### *indicator\_result\_specification*

```
int *indicator_result,
```

## Table Function Syntax Elements

### *function\_name*

Pointer to a C string whose value is the function name in the CREATE FUNCTION definition.

The function can use this name to build error messages.

***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE FUNCTION definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is the C type in `sqltypes_td.h` that corresponds to the SQL data type of *input\_parameter*.

***result\_specification***

Result row argument corresponding to one in the CREATE TABLE or REPLACE TABLE statement.

For a table function with fixed row result specification:

- The table has a *result\_specification* for each column in the RETURNS TABLE clause of the CREATE TABLE statement.
- Each *type* is the C type in `sqltypes_td.h` that matches the SQL data type of the corresponding column in the RETURNS TABLE clause of the CREATE TABLE statement.
- Each *result* is a pointer to a data area big enough for a value of the SQL data type of the corresponding column in the RETURNS TABLE clause of the CREATE TABLE statement.

For a table function with dynamic row result specification:

- The table has a *result\_specification* for each column in the RETURNS TABLE VARYING COLUMNS clause of the CREATE TABLE statement.
- Each *result* is a void pointer, because the data types of the result row arguments are unknown until function invocation.

While running, the table function can get the data types of the result rows by calling the library function `FNC_TblGetColDef`.

- Each *result* is a pointer to a data area big enough for a value of the SQL data type of the corresponding column in the RETURNS TABLE clause of the CREATE TABLE statement.

***indicator\_parameter\_specification***

[Optional] Indicator parameter corresponding to an input parameter.

Each *input\_parameter\_specification* must have a corresponding *indicator\_parameter\_specification*. The input parameters and indicator parameters must be in the same order.

If the value of *indicator\_parameter* is -1, the value of the corresponding *input\_parameter* is null.

If the value of *indicator\_parameter* is 0, the value of the corresponding *input\_parameter* is a non-null value.

### ***indicator\_result\_specification***

[Optional] Indicates whether the table function returns the result corresponding *indicator\_parameter*. Each *indicator\_parameter\_specification* must have a corresponding *indicator\_result\_specification*. The indicator parameters and indicator result parameters must be in the same order.

If the value of *indicator\_result* is -1, the table function does not return the result the corresponding *indicator\_parameter*.

If the value of *indicator\_result* is 0, the table function returns the result the corresponding *indicator\_parameter*.

For a table function with dynamic result row specification, an *indicator\_result* value of -1 means the corresponding result argument was omitted in the SELECT statement that invoked the table function.

If the table function returns NULL for a result argument, the corresponding *indicator\_result* must have the value -1.

### ***sqlstate***

Pointer to a six-character C string that indicates the SQLSTATE value—success, exception, or warning. The first five characters are ASCII and the sixth is the C null character. The string is initialized to '00000', which indicates success.

For more information on SQLSTATE values, see [Returning SQLSTATE Values](#).

### ***m***

Number of characters in the function name in the CREATE FUNCTION definition. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

### ***specific\_function\_name***

Pointer to a C string whose value is the name of the external function being invoked.

If the CREATE FUNCTION statement includes the SPECIFIC clause, *specific\_function\_name* is the name in the SPECIFIC clause; otherwise, *specific\_function\_name* is the same as *function\_name*.

The function can use this name to build error messages.

*l*

Number of characters in the name of the external function. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a function name.

***error\_message***

Pointer to a C string whose value is the the error message text.

*p*

Number of characters in the error message text. The maximum value is 256.

**Related Information:**

[Argument and Result Behavior](#)

**Example: Table Function with Fixed Result Row Specification**

Here is an example of how to declare a table function:

```

/***** C source file name: store_data.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void get_store_data( INTEGER *filetoread, /* input argument */
                    INTEGER *storeno, /* output argument */
                    INTEGER *itemno, /* output argument */
                    INTEGER *quantsold, /* output argument */
                    int *filetoreadIsNull,
                    int *storenoIsNull,
                    int *itemnoIsNull,
                    int *quantsoldIsNull;
                    char sqlstate[6],
                    SQL_TEXT extname[129],
                    SQL_TEXT specific_name[129],
                    SQL_TEXT error_message[257] )
{
    ...
}

```

The corresponding CREATE FUNCTION statement that installs the table function on the server looks like this:

```

CREATE FUNCTION getStoreData
  (FileToRead INTEGER)
RETURNS TABLE
  (StoreNo    INTEGER
   ,ItemNo    INTEGER
   ,QuantSold INTEGER)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getstoredata!udfsrc/store_data.c!F!get_store_data'
PARAMETER STYLE SQL;

```

Here is an example of an INSERT ... SELECT statement that invokes the table function in the FROM clause:

```

INSERT INTO Sales_Table
SELECT *
FROM TABLE (getStoreData(9005)) AS tf;

```

## Example: Table Function with Dynamic Result Row Specification

Here is an example of how to declare a table function with dynamic result row specification:

```

/***** C source file name: store_data.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void get_store_data( INTEGER *filetoread,    /* input argument */
                   void *out1,             /* output argument */
                   void *out2,             /* output argument */
                   void *out3,             /* output argument */
                   void *out4,             /* output argument */
                   int *filetoreadIsNull,
                   int *out1IsNull,
                   int *out2IsNull,
                   int *out3IsNull;
                   int *out4IsNull;
                   char sqlstate[6],
                   SQL_TEXT extname[129],
                   SQL_TEXT specific_name[129],
                   SQL_TEXT error_message[257] )
{

```

```
...
}
```

The corresponding CREATE FUNCTION statement that installs the table function on the server looks like this:

```
CREATE FUNCTION getStoreData
  (FileToRead INTEGER)
RETURNS TABLE VARYING COLUMNS (4)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getstoredata!udfsrc/store_data.c!F!get_store_data'
PARAMETER STYLE SQL;
```

Here is an example of an INSERT ... SELECT statement that invokes the table function in the FROM clause:

```
INSERT INTO Sales_Table
SELECT *
FROM TABLE (getStoreData(9005)
  RETURNS (Store INTEGER, Item INTEGER, Quantity INTEGER)) AS tf;
```

## Constant Mode Table Function Body

This section provides guidelines on how to implement the body of a table function that the SELECT statement invokes with constant expression input arguments.

For example, the following statement invokes table\_function\_1 with a constant argument:

```
SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;
```

---

### Note:

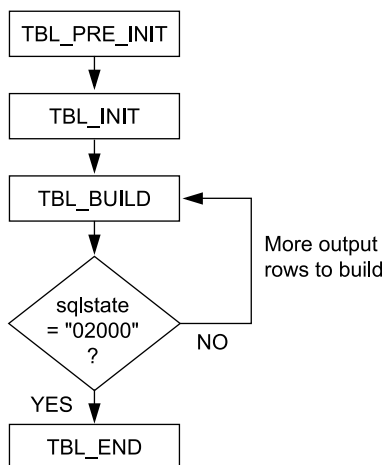
Although this section discusses how to implement a table function that only handles constant expression input arguments, you can design a table function that can handle constant and variable input arguments. For details on how to implement a variable mode table function, see [Variable Mode Table Function Body](#).

---

## Phases of a Constant Mode Table Function

Vantage invokes a constant mode table function repeatedly, allowing the function to pass through different phases where it can perform initialization, connect to external objects, build output rows, close open connections, and clean up. The function uses the `FNC_GetPhase` library function to determine the phase in which it was called and what action to take.

Here are the phases that the function passes through:



For more information on implementing each phase of a constant mode table function, see [Implementation Guidelines](#).

## Implementation Guidelines

Here are the basic steps you take to define a table function that is invoked with constant expression input arguments:

1. Define the `SQL_TEXT` constant.  
For more information, see [SQL\\_TEXT Definition](#).
2. Include the `sqltypes_td.h` header file.  
For more information, see [Header Files](#).
3. Include other header files that define macros and variables that the function uses.
4. Define the function parameter list in the order that the `CREATE FUNCTION` statement specifies the parameters.  
For more information, see [Table Function Parameter List](#).
5. If the table function is defined with dynamic result row specification, call the `FNC_TblGetColDef` library function to get the actual number and data types of the result row arguments that the table function must return.

6. Call the `FNC_GetPhase` library function and verify that the mode is `TBL_MODE_CONST`, indicating that the table function was invoked with constant expression input arguments.
7. Use the value that the `FNC_GetPhase` library function returns for the `FNC_Phase` argument to determine the phase in which the function was called and what action to take.

IF the value is ...	THEN the table function ...
<code>TBL_PRE_INIT</code>	<p>must decide whether it should be the controlling copy of all table functions running on other AMP vprocs.</p> <p>If the function wants to provide control context to all other copies of the table function, the function must call <code>FNC_TblControl</code>.</p> <p>If the function does not want to be the controlling copy of the table function, or if the function is designed without the need for a controlling function, the function can simply return and do nothing during this phase.</p> <p>All copies of the table function must complete this phase before any copy continues to the <code>TBL_INIT</code> phase.</p>
<code>TBL_INIT</code>	<p>should open any connections to external objects, such as files, if there is a need to do so. Any copy of the function that does not want to participate further must call <code>FNC_TblOptOut</code>. After the function returns, it will not be called again.</p> <p>All copies of the table function must complete this phase before any copy continues to the <code>TBL_BUILD</code> phase.</p>
<code>TBL_BUILD</code>	<p>should take one of the following actions:</p> <p>If the function has a row to build, then...</p> <ol style="list-style-type: none"> <li>a. Build an output row by filling out each result argument whose corresponding <i>indicator_result</i> argument has an input value of 0 (not NULL).</li> <li>b. Set the <i>indicator_result</i> arguments for the result values. <ul style="list-style-type: none"> <li>If the result argument is NULL, then set the corresponding <i>indicator_result</i> argument to -1.</li> <li>If the result argument is not NULL, then set the corresponding <i>indicator_result</i> argument to 0.</li> </ul> </li> </ol> <p>The function remains in the <code>TBL_BUILD</code> phase.</p> <p>If the function has no row to build, then set the <code>sqlstate</code> argument to "02000" to indicate no data. The function continues to the <code>TBL_END</code> phase.</p>
<code>TBL_END</code>	<p>should close all external connections and release any scratch memory it might have allocated. The table function is not called again after it returns from this phase.</p> <p>The controlling copy of the table function, if one exists, is called with this phase after all other copies of the table function have completed this phase, allowing the controlling function to do any final cleanup or notification to the external world.</p>
<code>TBL_ABORT</code>	<p>is being aborted and should close all external connections and release any previously-allocated memory. A function can be called at any time with this phase, which is only entered when one of the table functions calls the library function <code>FNC_TblAbort</code>. It is not entered when the function is aborted for an external reason, such as a user abort.</p>

8. Implement the function and set the results to the appropriate value.
9. If the function detects an error, set the:



- sqlstate argument to an SQLSTATE exception or warning condition before exiting.  
For more information, see [Returning SQLSTATE Values](#).
- error\_message string to the error message text. The characters must be inside the LATIN character range. The string is initialized to a null-terminated string on input.

## Design Considerations

A constant mode table function is sent to all AMP vprocs and each copy is passed the same input arguments. Each copy is called repeatedly until the function indicates it is finished.

During the design of a table function, you must determine whether it makes sense for all table function loops on all AMP vprocs to participate in the processing.

For example, a typical constant mode table function most likely reads data from outside the database to produce result rows. If the external data is only available on one node, it might be practical to have only one function copy on one vproc do anything useful. On the other hand, if the external data is available on each node, then perhaps it can be read from all AMP vprocs.

IF you want ...	THEN ...
all copies of the table function to participate in the processing	<p>use the following code excerpt as a guideline to implement your function.</p> <pre> FNC_Phase  Phase;  if ( FNC_GetPhase(&amp; Phase) != TBL_MODE_CONST) {     /* set sqlstate to an error and return */     strcpy(sqlstate, "U0005");     return; }  /* depending on the phase decide what to do */ switch(Phase) {     case TBL_PRE_INIT:     {         break;     }     case TBL_INIT:     {         /* Open any files here. */         ...          break;     }     case TBL_BUILD:     {         /* Read from files and build the result row here.      */         /* On EOF, set sqlstate to "02000" (no row to build). */         ...         break;     } } </pre>

IF you want ...	THEN ...
	<pre> case TBL_END: {     /* Everyone done. Close files. */     ...     break; } case TBL_ABORT: {     /* A copy called FNC_TblAbort. Close files. */     ...     break; } } </pre>
<p>a table function that can run on any AMP and only needs one copy to participate</p>	<p>call the FNC_TblFirstParticipant library function from all copies of the table function. The first copy to make the call is the copy that participates. All other copies must call FNC_TblOptOut and return.</p> <p>Use the following code excerpt as a guideline to implement your function:</p> <pre> FNC_Phase  Phase;  if ( FNC_GetPhase(&amp; Phase) != TBL_MODE_CONST) {     /* set sqlstate to an error and return */     strcpy(sqlstate, "U0005");     return; }  /* depending on the phase decide what to do */ switch(Phase) {     case TBL_PRE_INIT:     {         switch (FNC_TblFirstParticipant() )         {             case 1: /* participant */                 return;             case 0: /* not participant */                 if (FNC_TblOptOut())                     strcpy(sqlstate, "U0006"); /* error */                 return;             default: /* -1 or other error */                 strcpy(sqlstate, "U0007");                 return;         }         break;     }     case TBL_INIT:     {         /* Open any files here. */         ...         break;     } } </pre>

IF you want ...	THEN ...
	<pre>     }     case TBL_BUILD:     {         /* Read from files and build the result row here. */         /* On EOF, set sqlstate to "02000" (no row to build). */         ...         break;     }     case TBL_END:     {         /* Everyone done. Close files. */         ...         break;     }     case TBL_ABORT:     {         /* A copy called FNC_TblAbort. Close files. */         ...         break;     } } </pre>
<p>one copy of the table function to be the controlling copy of all other copies running on all other AMP vprocs</p>	<p>call the FNC_TblControl library function to designate a copy of the table function as the controlling copy. Distribute data to other table function copies by calling FNC_TblAllocCtrlCtx to allocate a control scratchpad.</p> <p>Use the following code excerpt as a guideline to implement your function. For the complete code example, see <a href="#">C Table Function</a>.</p> <pre> typedef struct {     unsigned short cntrl_fnc_AMP     int            qfd;     ... } ctrl_ctx; ctrl_ctx *options; FNC_Phase Phase; AMP_Info_t *LocalConfig;  if ( FNC_GetPhase(&amp; Phase) != TBL_MODE_CONST) {     /* Set sqlstate to an error and return. */     strcpy(sqlstate, "U0005");     return; }  /* Depending on the phase decide what to do. */ switch(Phase) {     case TBL_PRE_INIT:     {         LocalConfig = FNC_AMPInfo();         /* Run controlling copy on the lowest AMP on the node. */         if (LocalConfig-&gt;LowestAMPOnNode) </pre>

IF you want ...	THEN ...
	<pre> { /* Run on the node that can access external file. */ if (access('filetoread')) { if ( FNC_TblControl() ) { /* Use scratchpad to distribute data to other */ /* function copies during the TBL_INIT phase. */ options = FNC_TblAllocCtrlCtx(sizeof(ctrl_ctx)); options-&gt;ctrl_fnc_AMP = LocalConfig-&gt;AMPId; ... } } } } case TBL_INIT: { /* Get the data from the controlling copy. */ options = FNC_TblGetCtrlCtx(); ...  break; } case TBL_BUILD: { /* Build result row or set sqlstate to "02000" if no data */ ... break; } case TBL_END: { /* Everyone done. Close files. */ /* Controlling copy must do extra cleanup. */ ... break; } case TBL_ABORT: { /* A copy called FNC_TblAbort. Close files. */ ... break; } } </pre>

## Variable Mode Table Function Body

This section provides guidelines on how to implement the body of a table function that the SELECT statement invokes using the columns from a derived table as input arguments.

For example, the following statement invokes `table_function_2` using `column_1` from the preceding derived table with the `t1` correlation name:

```
SELECT *
FROM ( SELECT column_1
      FROM table_1
      WHERE column_2 > 65 ) AS t1,
      TABLE (table_function_2(t1.column_1)) AS t2 (c1, c2, c3)
WHERE t1.column_3 = t2.c3;
```

---

**Note:**

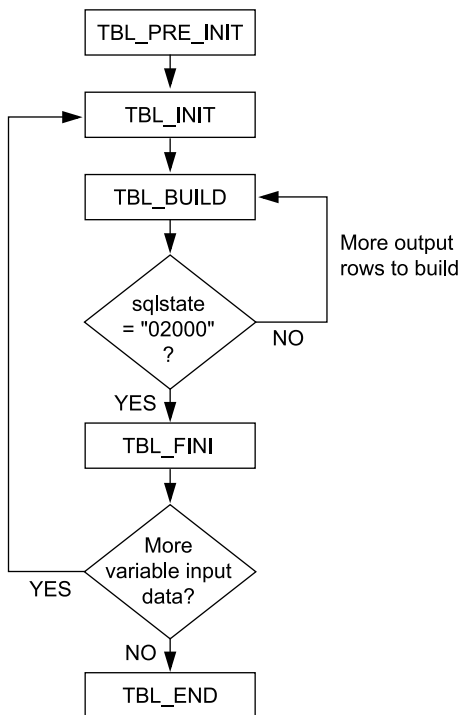
Although this section discusses how to implement a table function that only handles variable input arguments, you can design a table function that can handle constant and variable input arguments. For more information on how to implement a constant mode table function, see [Constant Mode Table Function Body](#).

---

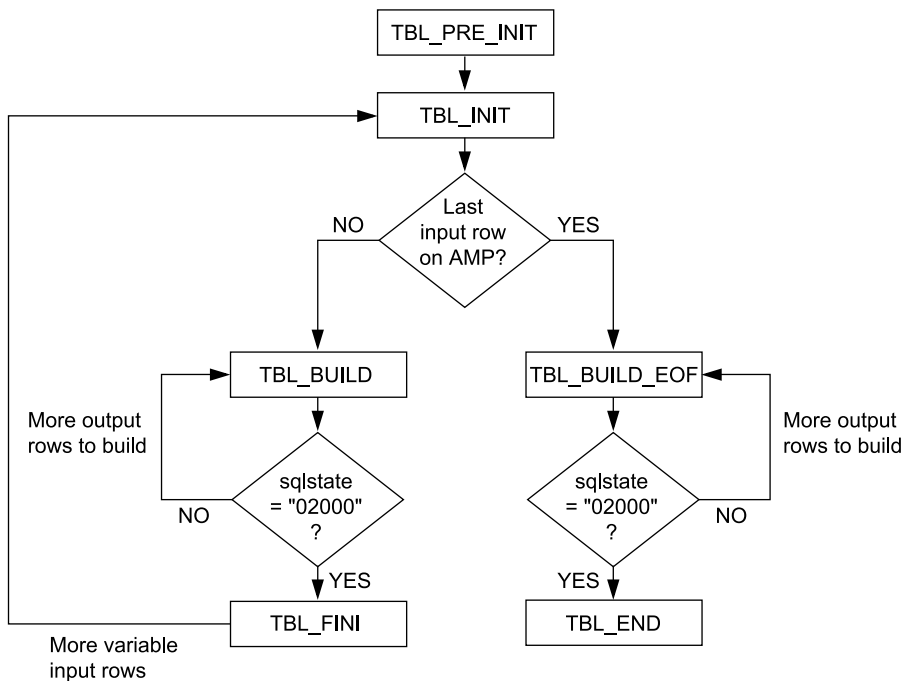
## Phases of a Variable Mode Table Function

Vantage invokes a variable mode table function repeatedly, allowing the function to pass through different phases where it initializes itself, connects to external objects, builds output rows, closes open connections, and cleans up. The function can use the `FNC_GetPhase` or the `FNC_GetPhaseEx` library function to determine which phase it is in and what action to take.

A function that uses the `FNC_GetPhase` library function passes through the following phases.



Alternatively, a table function that needs to know when it is passed in the last qualified row on an AMP (perhaps because the function does not return a row until it has processed all of the input rows) can use the FNC\_GetPhaseEx function with the TBL\_LASTROW option and pass through the following phases.



For more information on implementing each phase of a variable mode table function, see [Implementation Guidelines](#).

## Implementation Guidelines

Here are the basic steps you take to define a table function that is invoked using the columns from a derived table as input arguments:

1. Define the SQL\_TEXT constant.

For more information, see [SQL\\_TEXT Definition](#).

2. Include the sqltypes\_td.h header file.

For more information, see [Header Files](#).

3. Include other header files that define macros and variables that the function uses.

4. Define the function parameter list in the order that the CREATE FUNCTION statement specifies the parameters.

For more information, see [Table Function Parameter List](#).

5. If the table function is defined with dynamic result row specification, call the FNC\_TblGetColDef library function to get the actual number and data types of the result row arguments that the table function must return.
6. Call the FNC\_GetPhase or FNC\_GetPhaseEx library function and verify that the mode is TBL\_MODE\_VARY, indicating that the table function was invoked using the columns from a derived table as input arguments.
7. Use the value returned by the FNC\_GetPhase or FNC\_GetPhaseEx library function for the FNC\_Phase argument to determine the phase in which the function was called and what action it should perform.

IF the value is ...	THEN the table function ...
TBL_PRE_INIT	is being called for the first time for all the rows that it will be called for. The input arguments to the function contain the first set of data. During this phase, the function has an opportunity to establish overall global context, but should not build any result row. The function continues to the TBL_INIT phase.
TBL_INIT	should open any connections to external objects, such as files, if there is a need to do so. The input arguments to the function contain the first set of data. During this phase, the function should not build any result row. The function continues to the TBL_BUILD phase.
TBL_BUILD	should take one of the following actions: <ul style="list-style-type: none"> <li>• If the function has a row to build, then build an output row by filling out each result argument whose corresponding <i>indicator_result</i> argument has an input value of 0 (not NULL). Set the <i>indicator_result</i> arguments for the result values as follows: <ul style="list-style-type: none"> <li>◦ If the result argument is NULL, then set the corresponding <i>indicator_result</i> argument to -1.</li> </ul> </li> </ul>

IF the value is ...	THEN the table function ...
	<ul style="list-style-type: none"> <li>If the result argument is not NULL, then set the corresponding <i>indicator_result</i> argument to 0.</li> </ul> <p>The function remains in the TBL_BUILD phase.</p> <ul style="list-style-type: none"> <li>If the function has no row to build, then set the <i>sqlstate</i> argument to "02000" to indicate no data.</li> </ul> <p>The function continues to the TBL_FINI phase.</p> <p>If using FNC_GetPhaseEx, the following actions are done depending on the option specified:</p> <ul style="list-style-type: none"> <li>If the TBL_NEWROW option is set, then call the table function with a new row with a phase of TBL_BUILD.</li> <li>If the TBL_NEWROWEOF option and EOF are set, then call the table function with a new row with a phase of TBL_BUILD.</li> <li>If the TBL_LASTROW option is set, the function remains in the TBL_BUILD phase until it is passed in the last set of data, where it continues to the TBL_BUILD_EOF phase.</li> </ul>
TBL_BUILD_EOF (this value is only returned by FNC_GetPhaseEx)	<p>is being called after the last input row on the AMP was passed in. The function has an opportunity to output a summary row of what it has collected in memory during previous calls when the phase was TBL_BUILD. The function should take one of the following actions.</p> <ul style="list-style-type: none"> <li>If the function has a row to build, then build an output row by filling out each result argument whose corresponding <i>indicator_result</i> argument has an input value of 0 (not NULL). Set the <i>indicator_result</i> arguments for the result values as follows: <ul style="list-style-type: none"> <li>If the result argument is NULL, then set the corresponding <i>indicator_result</i> argument to -1.</li> <li>If the result argument is not NULL, then set the corresponding <i>indicator_result</i> argument to 0.</li> </ul> </li> </ul> <p>The function remains in the TBL_BUILD_EOF phase.</p> <ul style="list-style-type: none"> <li>If the function has no row to build, then set the <i>sqlstate</i> argument to "02000" to indicate no data.</li> </ul> <p>The function continues to the TBL_END phase.</p>
TBL_FINI	<p>should close any connections, such as file handles, that were opened during the TBL_INIT phase.</p> <p>If there is more variable input data, the function returns to the TBL_INIT phase. Otherwise, the function continues to the TBL_END phase.</p>
TBL_END	<p>should close all external connections and release any scratch memory it might have allocated. The table function is not called again after this phase.</p>
TBL_ABORT	<p>is being aborted and should close all external connections and release any previously-allocated memory. A function can be called at any time with this phase, which is only entered when one of the table functions calls the library function FNC_TblAbort. It is not entered when the function is aborted for an external reason, such as a user abort.</p>

8. If the function detects an error, set the:

- sqlstate* argument to an SQLSTATE exception or warning condition before the function exits.



For more information, see [Returning SQLSTATE Values](#).

- `error_message` string to the error message text. The characters must be inside the LATIN character range. The string is initialized to a null-terminated string on input.

In general, you should understand the following row states:

- When the first row is passed, during the TBL\_PRE\_INIT phase.
- When a row is being returned, as indicated by the value of `sqlstate` in the TBL\_BUILD phase.
- When you want a new row. You can define this using the TBL\_NEWROW or TBL\_NEWROWEOF options of FNC\_GetPhaseEx.
- When the end of file has been encountered. You can determine this using the TBL\_LASTROW option of FNC\_GetPhaseEx.

## Improving Performance with Phase Reductions

The FNC\_GetPhaseEx options give you more control of table phase transitions when developing variable mode table functions. You can use them to reduce the number of phase transitions required during execution of a table function, thus reducing the number of UDF invocations and improving table function performance.

The following table compares the number of phase transitions required for each row if you do not use the FNC\_GetPhaseEx options and if you do use the various options. The phases are P (TBL\_PRE\_INIT), I (TBL\_INIT), B (TBL\_BUILD), B EOF (TBL\_BUILD signalling EOF, not TBL\_BUILD\_EOF), F (TBL\_FINI), and E (TBL\_END). X is the scale factor of input to output rows.

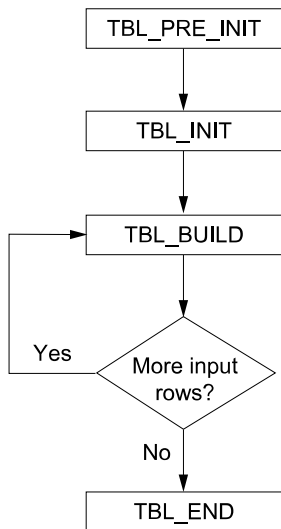
Processing Mode	Required phases if the FNC_GetPhaseEx options are not used	Required phases if the specified FNC_GetPhaseEx options are used
1:1 (one row in : one row out)	I, B, B EOF, F	B, if the TBL_NEWROW option is set.
1:M (one row in : many rows out)	I, B*X, B EOF, F	B * X, if the TBL_NEWROWEOF option is set.
M:1 (many rows in : one row out)	I, B EOF, F	B * X, if the TBL_NEWROW   TBL_LASTROW options are set.

For example:

In a 1:1 processing mode, use the TBL\_NEWROW option to get a new row on every TBL\_BUILD call.

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_NEWROW);
```

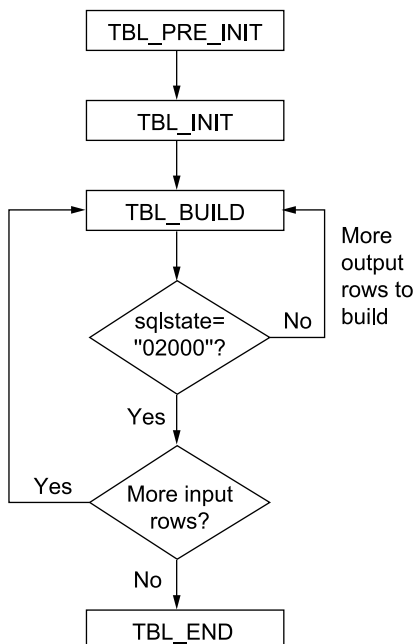
In this case, the function passes through the following phases.



In a 1:M processing mode, use the TBL\_NEWROWEOF option to get a new row when EOF is signaled.

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_NEWROWEOF);
```

In this case, the function passes through the following phases.



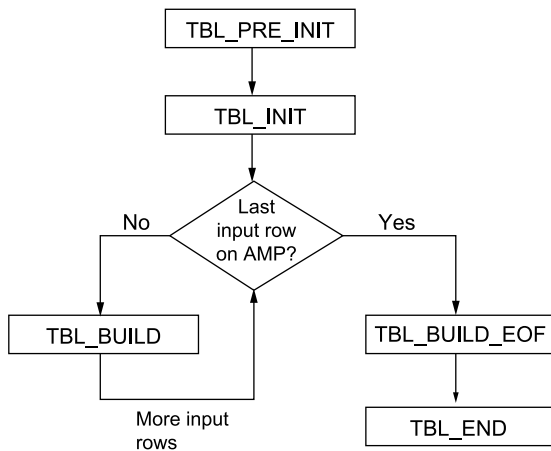
In a M:1 processing mode, you can use:

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_LASTROW | TBL_NEWROW);
```

In a combined M:1 and 1:M processing mode, you can use the following that does not pass a new row until EOF:

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_LASTROW | TBL_NEWROWEOF);
```

In this case, the function passes through the following phases.



For more information, see [FNC\\_GetPhaseEx](#).

## Design Considerations

The AMP vprocs that participate are those containing rows pertaining to the input data provided to the function from the correlated table name specified prior to the table function in the SELECT statement. The assumption is that the input argument determines what to process. Each copy of the function will be called with the same input data repeatedly until the function returns with the no more data condition.

Although it might be possible to read additional data from outside the database in this mode, it probably will not work. This is because the participating AMP vprocs are determined based on where the selected database rows reside. It might not be on all AMP vprocs.

Use the following code excerpt as a guideline to implement your function using FNC\_GetPhase:

```
typedef struct {
    scratch_XML xml_ctx;
} local_ctx;

FNC_Phase Phase;
local_ctx *state_info;

if ( FNC_GetPhase(& Phase) != TBL_MODE_VARY)
{
```

```

    /* Set sqlstate to an error and return. */
    strcpy(sqlstate, "U0005");
    return;
}

/* depending on the phase decide what to do */
switch(Phase)
{
    case TBL_PRE_INIT:
    {
        /* Allocate scratchpad to retain data between iterations. */
        state_info = FNC_TblAllocCtx(sizeof(local_ctx));
        break;
    }
    case TBL_INIT:
    {
        /* Get scratchpad. */
        state_info = FNC_TblGetCtx;
        /* Preprocess data here. */
        ...
        break;
    }
    case TBL_BUILD:
    {
        /* Get scratchpad and build the result row here. */
        state_info = FNC_TblGetCtx;
        ...
        /* Or, if no more rows to build, set sqlstate to "02000". */
        strcpy(sqlstate, "02000");
        ...
        break;
    }
    case TBL_FINI:
    {
        /* Reset for the next set of data. */
        state_info = FNC_TblGetCtx;
        ...
        break;
    }
    case TBL_END:
    {
        /* Everyone done. */
        ...
        break;
    }
}

```

```

    }
}

```

If your table function does not build a result row until after it receives all of the input rows for the AMP on which it runs, you can use `FNC_GetPhaseEx` with the `TBL_LASTROW` option instead of `FNC_GetPhase`. This will build the row during the `TBL_BUILD_EOF` phase. Use the following code excerpt as a guideline to implement your function using `FNC_GetPhaseEx`.

```

typedef struct {
    scratch_XML xml_ctx;
} local_ctx;

FNC_Phase Phase;
local_ctx *state_info;

if ( FNC_GetPhaseEx(& Phase, TBL_LASTROW) != TBL_MODE_VARY)
{
    /* Set sqlstate to an error and return. */
    strcpy(sqlstate, "U0005");
    return;
}

/* depending on the phase decide what to do */
switch(Phase)
{
    case TBL_PRE_INIT:
    {
        /* Allocate scratchpad to retain data between iterations. */
        state_info = FNC_TblAllocCtx(sizeof(local_ctx));
        break;
    }
    case TBL_INIT:
    {
        /* Get scratchpad. */
        state_info = FNC_TblGetCtx;
        /* Preprocess data here. */
        ...
        break;
    }
    case TBL_BUILD:
    {
        /* Get scratchpad and add data here. */
        state_info = FNC_TblGetCtx;
        ...
        /* Set sqlstate to "02000" to indicate not to output result rows here. */
        strcpy(sqlstate, "02000");
        ...
        break;
    }
    case TBL_BUILD_EOF:
    {
        /* Get scratchpad and build result row here. */
        state_info = FNC_TblGetCtx;
        ...
        /* Or, if no more rows to build, set sqlstate to "02000". */
        strcpy(sqlstate, "02000");
        ...
        break;
    }
    case TBL_FINI:
    {
        /* Reset for the next set of data. */

```

```

        state_info = FNC_TblGetCtx;
        ...
        break;
    }
    case TBL_END:
    {
        /* Everyone done. */
        ...
        break;
    }
}

```

If you want to get a new row on each table function invocation, use `FNC_GetPhaseEx` with the `TBL_NEWROW` option. The following code excerpt shows an example of this usage.

```

{
    FNC_Phase  Phase;

    /* Make sure the function is called in the supported context. */
    /* Only ask for the phase on each row. */
    switch (FNC_GetPhaseEx(&Phase, TBL_NOOPTIONS))
    {
        case TBL_MODE_CONST:
            strcpy(sqlstate, "U0005");
            strcpy((char *) errmsg, "Table function being called in unsupported mode.");
            return;
        case TBL_MODE_VARY:
            switch(Phase)
            {
                case TBL_PRE_INIT:
                    /* Ask for a new row on each call to build */
                    FNC_GetPhaseEx(&Phase, TBL_NEWROW);
                    break;
                case TBL_INIT:
                    break;
                case TBL_BUILD:
                    *out1 = *in1;
                    strcpy((char*)out2, (char*)in2);
                    break;
                case TBL_FINI:
                    break;
                case TBL_END:
                    break;
            }
            break;
    }
}

```

Note that the above sample code is simply returning the values of the input arguments as output columns, so there is no need for a context object to retain status information between iterations of the table UDF. If you need to use a context object to retain status information between iterations of a table UDF, you must include code for defining, allocating, setting and getting the context object in the appropriate phases accordingly.

## Table Operators

You can create user-defined table operators, which accept one or more tables or table expressions as input and generate a table as output. A table operator is a form of UDF that can only be specified in the FROM clause of a SELECT statement.

You can define a table operator that reads input tables, performs operations on the tables such as partitioning or aggregation, then writes output rows. The table operator can accept an arbitrary row format for each input table or table expression and based on the operation and input row types, it can produce an arbitrary output row format.

---

### Note:

Table functions and table operators cannot execute against fallback data when an AMP is down. Once the AMP returns to service, the query can be submitted.

---

## Differences Between Table Functions and Table Operators

- The inputs and outputs for table operators are a set of rows (a table) and not columns. The default format of a row is IndicData.
- In a table function, the row iterator is outside of the function and the iterator calls the function for each input row. In the table operators, the operator writer is responsible for iterating over the input and producing the output rows for further consumption. The table operator itself is called only once. This reduces per row costs and provides more flexible read/write patterns.
- The operator writer has the option of establishing the output characteristics via a user-defined contract function. The contract function is invoked at runtime when the table operator is being parsed. Therefore, the operator has full flexibility to determine the output from the input.
- The table operators can use custom argument clauses to make them polymorphic.

## Table Operator Basic Structure

The table operator must perform the iteration over the input rows; therefore, you should structure the operator as follows:

- Open the input stream.
- Process the rows by iteration.
- Close the stream.

You define a table operator by specifying the SQLTABLE parameter style in the CREATE FUNCTION statement. With this parameter style, a table operator is passed an input row in the IndicData format and is expected to return the output row in the IndicData format as default.

For input streams:

- You can open and close an input stream multiple times within a table operator invocation. Each open resets the read position to the start of the stream.

- You can optionally decide to never open the input stream, read only a subset of the input rows or open but never read an input stream.
- You cannot close an input stream that is not open, open an already open input stream, or read past the end of file.

For output streams:

- You can open and close an output stream a single time within a table operator invocation.
- You can optionally decide to never open the output stream or open but never write the output stream.
- You cannot close an output stream that is not open, open an already open output stream, or close and reopen an output stream.

The completion determination mechanism is when the table operator exits.

If the PARTITION BY clause was specified, then opening and closing a stream refers to the rows within the partition. For example, End of File occurs after the last row in the partition was read. If PARTITION BY was not specified, then opening and closing a stream refers to the rows within the AMP.

The following SQLTABLE iterator functions are available for table operator writers:

- FNC\_TblOpGetStreamCount
- FNC\_TblOpOpen
- FNC\_TblOpRead
- FNC\_TblOpGetAttributeByNdx
- FNC\_TblOpBindAttributeByNdx
- FNC\_TblOpWrite
- FNC\_TblOpClose

For more information about these functions, see [Table Operator Interface](#).

## Contract Function

You can write a user-defined contract function which determines the output table row format of your table operator. The contract function is invoked at runtime when the table operator is being parsed.

The contract function is similar to a standard scalar UDF. The function takes no input parameters and returns an integer SUCCESS or FAILURE value.

The contract function can access table operator input information using FNC functions and set the return row format using an FNC function. The contract function context is accessible by the table operator.

The contract function can use FNC functions to access the following metadata:

- Input parameter metadata such as the number of columns, data types and name information.
- Explicitly defined output row metadata such as the number of columns, data types and name information.
- The name of the function being called.
- Metadata associated with the optional Custom clauses.
- Metadata for the optional HASH BY clause.



- Metadata for the optional LOCAL ORDER BY clause.

The contract function can use FNC functions to set the following metadata associated with the table operator:

- Output parameter metadata such as the number of columns, data types and name information.
- Optional structure that is created at contract time and passed to the table operator at runtime.
- Optional specification of the row and field format.
- Optional specification of the HASH BY clause.
- Optional specification of the LOCAL ORDER BY clause.

For details about the FNC functions that the contract function can use, see [Table Operator Interface](#).

You associate a contract function with a table operator by using the RETURNS TABLE clause of the CREATE FUNCTION statement.

For example, if you have a table operator defined as:

```
void udaggregation ()
{
    ...
}
```

And you have a contract function defined as:

```
void udaggregation_contract (
    INTEGER *result,
    int *indicator_Result,
    char sqlstate[6],
    SQL_TEXT extname[129],
    SQL_TEXT specific_name[129],
    SQL_TEXT error_message[257])
{
    ...
}
```

Then you associate the contract function with the table operator by specifying the name of the contract function in the RETURNS TABLE clause in the SQL definition of the table operator as follows:

```
REPLACE FUNCTION udaggregation ()
    RETURNS TABLE VARYING USING FUNCTION udaggregation_contract
    LANGUAGE C
    NO SQL
    PARAMETER STYLE SQLTABLE
    EXTERNAL NAME 'CS!udaggregation!udaggregation.c!F!udaggregation';
```

**Note:**

If the code for the table operator and the code for the contract function are in separate source files, the CREATE FUNCTION statement must reference both source files.

The maximum length of the contract function name is CHAR(30) CHARACTER SET LATIN. The CREATE FUNCTION statement fails if the contract function name exceeds this limit after the string is converted into LATIN for storage in the Data Dictionary.

Once created, you cannot drop a contract function directly. When the table operator is dropped, the contract function is also dropped. Similarly, you cannot alter a contract function directly. When the table operator is altered to run in a different mode, the contract function is also altered. When the table operator is recompiled, the contract function is also recompiled.

You cannot rename the table operator or the contract function.

For a C code example of a table operator and contract function, see [C Table Operator](#).

## C/C++ Table Operator Metadata Mapping

Data type metadata, including UDT and CDT metadata, can be passed to a table operator contract function.

Note the difference between the terms *mapping* and *transformed* type as follows:

### Mapping Type

The External Type code that is used in the contract function to identify the data type of a column.

### Transform Type

The predefined data type that a UDT or CDT value is converted to by invoking the transform function.

By default, all of the UDT or CDT data values are passed in their default transform form to the operator. This may correspond to either the SQL transform type of the UDT or another predefined type. The behavior is explicitly determined for each individual UDT or CDT type. For Java table operators, an option is provided to pass some UDTs and CDTs in an untransformed or atomized form.

Each of the UDT or CDT type is also mapped to an External Type code in the table operator contract function. Additional information about each type is passed in the UDT\_BaseInfo\_t structure for C/C++ table operators or the com.teradata.fnc.runtime.UDTBaseInfo class for Java table operators.

The following table shows the metadata mapping in the contract function from Teradata SQL type to the External Type code for UDTs and CDTs.

SQL UDT or Complex Data Type	External Type Code
ARRAY/VARRAY	ARRAY_DT
DATASET in the Avro storage format	DATASET_AVRO_DT

SQL UDT or Complex Data Type	External Type Code
DATASET in the CSV storage format	DATASET_CSV_DT
Geospatial – MBB	MBB_DT
Geospatial – MBR	MBR_DT
Geospatial – ST_Geometry	ST_GEOMETRY_DT
JSON	JSON_DT
Period type	PERIOD_DT
UDT (Distinct)	UDT_DT
UDT (Structured)	UDT_DT
XML	XML_DT

## Passing UDT and CDT Columns To and From a Table Operator

The contract function for a table operator describes the input and output columns to be passed and retrieved. The following sections provide detail on how UDT and CDT columns may be passed to and from a table operator.

## Retrieving UDT and CDT Input Metadata

In the contract function, an External Type code referring to the UDT or CDT is passed in the `parm_tx` structure for the input column.

The contract function retrieves the input column metadata by calling the `FNC_TblOpGetColDef` function. This function returns the structure `FNC_TblOpColumnDef_t` which contains a field of type `parm_tx` for each input column. The `datatype` field of the `parm_tx` structure identifies a UDT or CDT column using an External Type code.

There are two ways to retrieve the UDT and CDT metadata:

- `FNC_TblOpColumnDef_t` structures can be passed to the `FNC_TblOpGetBaseInfo` function, which fills out an array of `UDT_BaseInfo_t` structures corresponding to each input column. The `FNC_TblOpGetBaseInfo` function can only be invoked on the PE in the contract function.
- You can use the `FNC_TblOpGetUDTMetadata` function which returns `UDT_BaseInfo_t` structures for one or more input or output columns. This function can be invoked on both the PE and AMP vprocs.

For structured UDTs, the fields of the `UDT_BaseInfo_t` structure may not be sufficient to provide all the metadata information. Because structured UDTs may have many attributes and may also contain an arbitrary level of nesting, metadata about the attributes of a structured UDT is retrieved using the `FNC_TblOpGetStructuredAttributeInfo` function. This function returns an array of `attribute_info_t` structures corresponding to all of the attributes in the structured UDT.

## Setting UDT and CDT Output Metadata

The contract function calls the `FNC_TblOpSetOutputColDef` function to set the metadata for a group of output columns. The metadata is saved in the structure `FNC_TblOpColumnDef_t`. The data type encoding and udt name for the column can identify the UDT or CDT type output column. This is all the information that is needed to set the output metadata for a UDT or CDT column.

### Related Information:

[C/C++ Table Operator Default Transform Behavior](#)

## Example: C Table Operator That Retrieves UDT Metadata

The is an example of a C table operator that uses `FNC_TblOpGetBaseInfo()` to retrieve UDT metadata.

```
/*
CREATE FUNCTION mift1()
RETURNS TABLE VARYING USING FUNCTION mift1_contract
SPECIFIC mift1
LANGUAGE C
NO SQL
NO EXTERNAL DATA
PARAMETER STYLE SQLTable
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CS!mift1!mift1.c!F!mift1'; */

#define SQL_TEXT Latin_Text
#include <stdio.h>
#include <sqltypes_td.h>
#define BufferSize (32*1024)
#define SetError(e, m) strcpy((char *)sqlstate, ( e )); strcpy((char
*)error_message, ( m ))

typedef struct
{
    int colcount;
    FNC_TblOpColumnDef_t *iCols;
    FNC_TblOpHandle_t *Handle;
    int is_eof;
    int dimension;
}InputInfo_t;

int mift1_contract(
    INTEGER *Result,
```

```

        int *indicator_Result,
        char sqlstate[6],
        SQL_TEXT extname[129],
        SQL_TEXT specific_name[129],
        SQL_TEXT error_message[257])
{
    FNC_TblOpColumnDef_t    *oCols;
    FNC_TblOpColumnDef_t    *iCols;
    Stream_Fmt_en    format;
    InputInfo_t *icolinfo;
    int incount, outcount, ocolcount;
    int i,j, totalcols;
    UDT_BaseInfo_t *baseInfos;
    char mycontract[] = "this is my contract... this is my contract... this is my
contract...";

    FNC_TblOpGetStreamCount(&incount, &outcount);
    if(incount == 0)
    {
        SetError("U0003", "mift1 requires number of input streams to be > 0.");
        return -1;
    }

    icolinfo = FNC_malloc (incount * sizeof(InputInfo_t));

    totalcols = 0;
    for(i=0; i < incount; i++)
    {
        icolinfo[i].colcount = FNC_TblOpGetColCount(i, ISINPUT);
        totalcols += icolinfo[i].colcount;

        icolinfo[i].iCols = FNC_malloc(TblOpSIZECOLDEF(icolinfo[i].colcount));
        TblOpINITCOLDEF(icolinfo[i].iCols, icolinfo[i].colcount);
        FNC_TblOpGetColDef(i, ISINPUT, icolinfo[i].iCols);
    }

    /* Allocate space for columns. */
    oCols = (FNC_TblOpColumnDef_t *)FNC_malloc( TblOpSIZECOLDEF(totalcols) );
    memset(oCols, 0 , TblOpSIZECOLDEF(totalcols) );
    oCols->num_columns = totalcols;
    oCols->length = TblOpSIZECOLDEF(totalcols) - (2 * sizeof(int)) ;
    TblOpINITCOLDEF(oCols, totalcols);
    ocolcount = 0;

```

```

/* Copy input columns to output columns. */
for(j=0; j < incount; j++)
{
    iCols = icolinfo[j].iCols;
    baseInfos = (UDT_BaseInfo_t *)FNC_malloc(iCols->num_columns *
sizeof(UDT_BaseInfo_t));
    /* Get base info */
    FNC_TblOpGetBaseInfo(iCols, baseInfos);

    for(i=0; i < iCols->num_columns; i++)
    {
        switch (iCols->column_types[i].datatype)
        {
            case DECIMAL1_DT:
            case DECIMAL4_DT:
            case DECIMAL8_DT:
                oCols->column_types[ocolcount].size.range.totaldigit
                    = iCols->column_types[i].size.range.totaldigit;
                oCols->column_types[ocolcount].size.range.fracdigit
                    = iCols->column_types[i].size.range.fracdigit;
                break;
            case DECIMAL2_DT:
                oCols->column_types[ocolcount].size.range.totaldigit= 5;
                oCols->column_types[ocolcount].size.range.fracdigit=
                    iCols->column_types[i].size.range.fracdigit;
                break;
            case TIME_DT:
            case TIMESTAMP_DT:
            case PERIOD_DT:
                oCols->column_types[ocolcount].size.precision =
                    iCols->column_types[i].size.precision;
                break;
            case INTERVAL_YEAR_DT:
            case INTERVAL_YTM_DT:
            case INTERVAL_MONTH_DT:
            case INTERVAL_DAY_DT:
            case INTERVAL_DTH_DT:
            case INTERVAL_DTM_DT:
            case INTERVAL_DTS_DT:
            case INTERVAL_HOUR_DT:
            case INTERVAL_HTM_DT:
            case INTERVAL HTS_DT:
            case INTERVAL_MINUTE_DT:
            case INTERVAL_MTS_DT:

```

```

    case INTERVAL_SECOND_DT:
        oCols->column_types[ocolcount].size.intervalrange =
            iCols->column_types[i].size.intervalrange;
        break;
    case JSON_DT:
        oCols->column_types[ocolcount].JSONStorageFormat = JSON_TEXT_EN;
        oCols->column_types[ocolcount].charset =
            iCols->column_types[i].charset;
        oCols->column_types[ocolcount].size.length =
            iCols->column_types[i].size.length;
        oCols->column_types[ocolcount].udt_indicator = 5;
        memcpy(oCols->column_types[ocolcount].udt_type,
            iCols->column_types[i].udt_type, FNC_MAXNAMELEN_EON);
        break;
    case ARRAY_DT:
        oCols->column_types[ocolcount].charset =
            iCols->column_types[i].charset;
        oCols->column_types[ocolcount].size.length =
            iCols->column_types[i].size.length;
        oCols->column_types[ocolcount].udt_indicator = 4;
        memcpy(oCols->column_types[ocolcount].udt_type,
            iCols->column_types[i].udt_type, FNC_MAXNAMELEN_EON);
        oCols->column_types[ocolcount].struct_num_attributes=
            iCols->column_types[i].struct_num_attributes;
        break;

    default:
        oCols->column_types[ocolcount].size.length =
            iCols->column_types[i].size.length;
        oCols->column_types[ocolcount].charset = LATIN_CT;
        break;
}
oCols->column_types[ocolcount].datatype =
    iCols->column_types[i].datatype;
oCols->column_types[ocolcount].period_et =
    iCols->column_types[i].period_et;
oCols->column_types[ocolcount].bytesize =
    iCols->column_types[i].bytesize;
ocolcount++;
}

FNC_free(baseInfos);
}

```

```

FNC_TblOpSetContractDef(mycontract, strlen(mycontract)+1);
/* Define output columns. */
FNC_TblOpSetOutputColDef(0, oCols);
format = INDICFMT1;
FNC_TblOpSetFormat("RECFMT", 0, ISINPUT, &format, sizeof(format));
FNC_TblOpSetFormat("RECFMT", 0, ISOUTPUT, &format, sizeof(format));

FNC_free(oCols);
for(i=0; i < incount; i++)
    FNC_free(icolinfo[i].iCols);

FNC_free(icolinfo);
*Result = 1;
}

void mift1()
{
    int i;
    FNC_TblOpColumnDef_t *iCols, *oCols;
    FNC_TblOpHandle_t *Handle,*OutHandle;
    Stream_Fmt_en format;
    int incount, outcount,j;
    InputInfo_t *icolinfo;
    int *Result;
    int allStreamsEOF = 0;
    LOB_CONTEXT_ID lobid;
    FNC_LobLength_t actlen;
    LOB_RESULT_LOCATOR lrl_a[32];
    int truncerr;
    BYTE Buffer[BufferSize];
    int foundrow = 0;
    int isfirsttime = 1;

    FNC_TblOpGetStreamCount(&incount, &outcount);
    icolinfo = FNC_malloc (incount * sizeof(InputInfo_t));

    for(i=0; i < incount; i++)
    {
        /* Get Column Count */
        icolinfo[i].colcount = FNC_TblOpGetColCount(i, ISINPUT);

        /* Get Column Definitions */
        icolinfo[i].iCols = FNC_malloc(TblOpSIZECOLDEF(icolinfo[i].colcount));
        TblOpINITCOLDEF(icolinfo[i].iCols, icolinfo[i].colcount);
    }
}

```



```

FNC_TblOpGetColDef(i, ISINPUT, icolinfo[i].iCols);

/* Get Handles */
icolinfo[i].Handle = (FNC_TblOpHandle_t *)FNC_TblOpOpen(i,'r',0);
icolinfo[i].dimension = FNC_TblOpIsDimension(i,ISINPUT);
icolinfo[i].is_eof = 0;
}
Result = FNC_malloc(incount * SIZEOF_INTEGER);
OutHandle = (FNC_TblOpHandle_t*)FNC_TblOpOpen(0,'w',0);

isfirsttime = 1;
while(1)
{
    allStreamsEOF = 1;
    int ocount;
    for(j=0; j < incount; j++)
    {
        if ( icolinfo[j].is_eof == 0)
            Result[j] = FNC_TblOpRead(icolinfo[j].Handle);

        if( Result[j] == TBLOP_EOF)
            icolinfo[j].is_eof = 1;
        else
            if( Result[j] == TBLOP_SUCCESS)
                icolinfo[j].is_eof = 0;

        allStreamsEOF = allStreamsEOF & icolinfo[j].is_eof;
    }

    if(allStreamsEOF)
        break;
    /* We will write the o/p if atleast one non-dimesion o/p found a row */
    foundrow = 0 ;
    if( isfirsttime == 1 )
    {
        for( j=0; j < incount; j++)
        {
            if (icolinfo[j].dimension == 0 && icolinfo[j].is_eof == 0)
            {
                foundrow = 1;
            }
        }
    }
    if ( isfirsttime == 1 && foundrow == 0)

```

```

        break;
    isfirsttime = 0;
    ocount = 0;
    for(j=0; j < incount; j++)
    {
        Handle = icolinfo[j].Handle;
        iCols = icolinfo[j].iCols;
        for (i=0; i < iCols->num_columns; i++)
        {
            switch (iCols->column_types[i].datatype)
            {
                case CLOB_REFERENCE_DT:
                case BLOB_REFERENCE_DT:
                case JSON_DT:
                case XML_DT:
                case ST_GEOMETRY_DT:

                    /* This is output code, as we only have 1 outputstream should
be 0 */

                    lrl_a[ocount] = FNC_LobCol2Loc(0, ocount);

                    if(icolinfo[j].is_eof == 0)
                    {
                        if (!(TBLOPISNULL(Handle->row->indicators,i)))
                        {
                            FNC_LobOpen_CL(Handle->row->columnptr[i], &lobid, 0,
0);

                            truncerr = 0;
                            while( FNC_LobRead(lobid, Buffer, BufferSize, &actlen)
==
                                0 && !truncerr)
                                truncerr=FNC_LobAppend (lrl_a[ocount], Buffer, actlen,
&actlen);

                            FNC_LobClose(lobid);
                        }
                    }

                    break;
                default:

                    if(icolinfo[j].is_eof == 0)
                    {
                        OutHandle->row->columnptr[ocount] = Handle->row-
>columnptr[i];

```

```

        OutHandle->row->lengths[ocount] = Handle->row->lengths[i];
    }

    break;
}
if (icolinfo[j].is_eof == 1 || TBLOPISNULL(Handle->row-
>indicators,i))
{
    TBLOPSETNULL(OutHandle->row->indicators,ocount);
}
ocount++;
}
}
FNC_TblOpWrite(OutHandle); // Writes current output row
}
for(i=0; i < incount; i++)
{
    FNC_free(icolinfo[i].iCols);
    FNC_TblOpClose(icolinfo[i].Handle);    // close all contexts
}
FNC_free(icolinfo);
FNC_free(Result);
FNC_TblOpClose(OutHandle); // close all contexts
}

```

## C/C++ Table Operator Default Transform Behavior

### Default Transform Behavior for UDT and CDT Data Values

The input UDT and CDT data values sent to the table operator are in their default transform form. This may correspond to either the SQL transform type of the UDT or another predefined type. The UDT and CDT values are converted to their corresponding transform type by invoking the default transform function defined for the UDT or CDT. The following table shows the transform types for UDT and CDT data types passed to a table operator.

This mode is well suited for a connector-type of operator (T2T or T2M) that does import and export of data values.

Note that some UDTs and CDTs (such as XML, ST\_Geometry, DATASET, and JSON) support multiple transform groups. However, for table operators, the CDT values will always be sent using the *default* transform type. For instance, a JSON data value will always be sent as a CLOB.

SQL UDT or Complex Data Type	SQL Data Type Mapping	Java Data Type
ARRAY/ VARRAY	VARCHAR (the defined transform type)	java.lang.String

SQL UDT or Complex Data Type	SQL Data Type Mapping	Java Data Type
BSON	BLOB or CLOB	<ul style="list-style-type: none"> <li>• java.sql.Blob</li> <li>• java.sql.Clob</li> </ul>
DATASET – Avro	BLOB	java.sql.Blob
DATASET – CSV	CLOB	java.sql.Clob
Geospatial – MBB	VARCHAR	java.lang.String
Geospatial – MBR	VARCHAR	java.lang.String
Geospatial – ST_Geometry	CLOB	java.sql.Clob
JSON	CLOB	java.sql.Clob
Period	VARCHAR	java.lang.String
UDT (Distinct)	Predefined data type that the UDT is based on	Primitive Java data type For example: <ul style="list-style-type: none"> <li>• int</li> <li>• short</li> <li>• java.lang.String</li> <li>• java.sql.Clob</li> <li>• java.sql.Blob</li> </ul>
UDT (Structured)	Predefined data type as defined in CREATE TRANSFORM	Primitive Java data type For example: <ul style="list-style-type: none"> <li>• int</li> <li>• short</li> <li>• java.lang.String</li> <li>• java.sql.Clob</li> <li>• java.sql.Blob</li> </ul>
XML	CLOB	java.sql.Clob

## Handling Multiple Input Streams

You can read from multiple tables (input streams) inside table operators using multiple ON clauses. This allows table operators to be applied to related groups of information derived from different data sets. Data from these multiple data sources can be processed within a single SQL/MR function.

You can use the AS *name* clause to associate an optional alias with each input stream. To retrieve the alias name associated with an input stream, use the FNC\_TblOpGetAsClauseName function.

The order of the ON clauses must match the stream numbers in the table operator implementation. The maximum number of input streams for a table operator is 16.

The ON clause can accept the following options:

- No partitioning or order by attributes
- PARTITION BY ANY
- PARTITION BY ANY ORDER BY *column\_list*
- LOCAL ORDER BY *column\_list*
- PARTITION BY *column\_list*
- PARTITION BY *column\_list* ORDER BY *column\_list*
- HASH BY *column\_list*
- HASH BY *column\_list* LOCAL ORDER BY *column\_list*
- DIMENSION
- DIMENSION ORDER BY *column\_list*

---

**Note:**

The following rules apply:

- You cannot specify ORDER BY as the only clause in the input table. It must be combined with a PARTITION BY [ANY] or DIMENSION clause.
  - You cannot use scalar subqueries in table operators with multiple ON clauses or ON clauses using PARTITION BY or HASH BY.
- 

You can use the following functions to access and set HASH BY, PARTITION BY and ORDER BY metadata.

- FNC\_TblOpGetHashByDef
- FNC\_TblOpGetCountHashByDef
- FNC\_TblOpGetLocalOrderByDef
- FNC\_TblOpGetCountLocalOrderByDef
- FNC\_TblOpSetHashByDef
- FNC\_TblOpSetLocalOrderByDef

To determine if the input to the table operator is DIMENSION input, use the FNC\_TblOpsDimension function.

For more information about the FNC functions that you can use with table operators, see [Table Operator Interface](#).

For more information on using the ON, AS, HASH BY, PARTITION BY, ORDER BY, or DIMENSION clauses with table operators, see the information about SELECT in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Cogroup and Multiple Input Table Operators

Cogroup is a join operation where input from multiple input streams are grouped together into a huge group as long as the input satisfies the PARTITION BY condition.

For each invocation of the table operator, only the qualifying input are allowed to participate. Qualifying input is input that have the same partition key values.

The table operator is not invoked when all of the PARTITION BY ANY and PARTITION BY key inputs are empty while the DIMENSION inputs are not empty. The table operator is invoked for every partition and it can return one or more rows for every invocation.

Cogroup ensures that table operators that handle multiple input streams will return consistent results on systems with different configurations where the number of AMPs differ. This functionality is enabled by default, but you can disable cogroup by calling `FNC_TblOpDisableCoGroup` in the contract function. However, if cogroup is disabled, multiple input table operators may return different results on systems with different configurations.

---

**Note:**

Cogroup is not supported for table operators with an input record format set to `IndicData` with no row or partition separator sentinels (format attribute "RECFMT" set to `INDICBUFFMT1`). With this indicator buffer format, the table operator is not invoked per partition which is required by cogroup.

---

## Custom Clause

You can specify Custom clauses when invoking a table operator to customize and make the operator polymorphic. Custom clauses are similar in principle to the way `GROUP BY` and `HAVING` clauses apply to aggregation. The input is a list of key-value pairs. The contract function can validate the Custom clauses and return a syntax error if the Custom clauses are invalid. The clauses can contain names and literal values. For the literal value custom argument clauses, the type information is interpreted using the current syntaxer rules. For example, `1` is a `BYTEINT` and `'1'` is a `VARCHAR(1) CHARACTER SET x`, where `x` is the character set of the creator of the table operator. You cannot use a Teradata reserved word for a key, and the total length of custom clauses must be at most 64K bytes.

You can use the following functions to access information related to the Custom clauses, such as Type and Value.

- `FNC_TblOpGetCustomKeyCount`
- `FNC_TblOpGetCustomKeyInfoOf`
- `FNC_TblOpGetCustomKeyInfoAt`
- `FNC_TblOpGetCustomValuesOf`

For details about these functions, see [Table Operator Interface](#).

## Running Table Operators on a Set of AMPs

You can run table operators on a set of AMPs based on a contiguous or sparse map that is associated with the table operator. The database distributes input rows to the AMPs in the specified or default map before running the table operator on the specified or default map if the input rows are not already distributed per that map.

For example, you may want to configure a few nodes with extra memory so that those nodes can run special table operator queries, such as SQL-H queries, SAS jobs, or R scripts. You can define a map that includes only the AMPs on the special nodes and then specify that map in a CREATE/REPLACE FUNCTION statement using the EXECUTE MAP clause.

You can also use the EXECUTE MAP clause when invoking a table operator so that the table operator will execute only on the AMPs specified in the map. This overrides any map and colocation name specified when the table operator was created.

For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## Related Information

FOR more information on ...	SEE ...
the SQL definition for a table operator	CREATE FUNCTION (Table Form) information in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
invoking a table operator in an SQL query	FROM clause of the SELECT statement in <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146.
the table operator interface functions available to table operator and contract function writers	<a href="#">Table Operator Interface</a> .
example C code for a table operator and contract function	<a href="#">C Table Operator</a> .

## Returning SQLSTATE Values

The parameter list of a UDF includes an output character string for returning the SQLSTATE result code value.

## C Data Type

The following table defines the C data type that corresponds to the SQLSTATE result code variable.

Result Code Variable	C Data Type
SQLSTATE	char sqlstate[6]

## SQLSTATE Values

The first five characters of the *sqlstate* output character string have a format of 'ccsss', where *cc* is the class and *sss* is the subclass. The last character of the string is a binary 0, or C string terminator. The following table shows some of the more useful valid settings that a UDF can return for the SQLSTATE result code.

Category	Class	Condition	Subclass	Sub-condition
Success	00	No error.	000	None.
	02	No data.	000	None.
Warning	01	Warning.	H xx	External routine warning where you choose the value of xx.
Exception	38	External routine exception.	000	None.
			002	Invalid return value for predicate.
	39	External routine invocation exception.	000	None.
			001	Invalid SQLSTATE returned. This error indicates the function returns a bad SQLSTATE.
			002	Null value not allowed.
	22	Data exception.	000	None.
			021	Character not in repertoire.
			008	Datetime field overflow.
			012	Division by zero.
			005	Error in assignment.
			022	Indicator overflow.
			015	Interval field overflow.
			018	Invalid datetime format.
			019	Invalid escape character.
			025	Invalid escape sequence.
			010	Invalid indicator parameter value.
			020	Invalid limit value.
			023	Invalid parameter value.
			002	Null value, no indicator parameter.
			003	Numeric value out of range.
			004	Null value not allowed.
			026	String data length mismatch.
			001	String data, right truncation.
			027	Trim error.



Category	Class	Condition	Subclass	Sub-condition
			024	Unterminated C string.
			00F	Zero-length C string.
	U0	User-defined data exception.	xxx	User-defined exception where you choose the value of xxx.

## Initial Value

The *sqlstate* output character string is initialized to '00000' (five zero characters), which corresponds to a success condition. Therefore, you do not have to set the value of the *sqlstate* output argument for a normal return.

## Display Format

If a UDF returns an SQLSTATE value other than success, a BTEQ session displays an error.

IF the SQLSTATE category is ...	THEN the display format is ...
not a warning	<pre>*** Failure 7504 in UDF/XSP  dbname.udfname: SQLSTATE  ccsss: &lt;text&gt;</pre> <p>where:</p> <ul style="list-style-type: none"> <li>7504 is the Teradata designated error code for user-defined functions, user-defined methods, and external stored procedures</li> <li><i>dbname.udfname</i> is the name of the UDF and the name of the database in which the UDF resides</li> <li><i>ccsss</i> is the value that the UDF sets the SQLSTATE output argument to, according to the table in <a href="#">SQLSTATE Values</a>.</li> <li><i>&lt;text&gt;</i> is the value of the error message output argument, if the UDF uses parameter style SQL</li> </ul>
warning	<pre>*** Warning: 7505 in UDF/XSP  dbname.udfname: SQLSTATE 01H xx: &lt;text&gt;</pre> <p>where:</p> <ul style="list-style-type: none"> <li>7505 is the Teradata designated warning code for user-defined functions, user-defined methods, and external stored procedures</li> <li><i>dbname.udfname</i> is the name of the UDF and the name of the database in which the UDF resides</li> <li>01H xx is the value that the UDF sets the SQLSTATE output argument to, according to values for the warning category in the table in <a href="#">SQLSTATE Values</a>.</li> <li><i>&lt;text&gt;</i> is the value of the error message output argument, if the UDF uses parameter style SQL</li> </ul>

## Value Returned to Stored Procedures

When a stored procedure control statement calls a UDF, the SQLSTATE result code value is the same value that the UDF sets, not a value of 'T7505' or 'T7504' as might be expected.

## Example: Returning the SQLSTATE Result Code Value

Consider the following standard deviation aggregate function:

```
void STD_DEV ( FNC_Phase      phase,
               FNC_Context_t *fctx,
               FLOAT          *x,
               FLOAT          *result,
               char            sqlstate[6] )
{
    ...
}
```

You can use the *sqlstate* argument to return the SQLSTATE result code value.

For example, if the value of the *phase* argument is not a valid aggregation phase, you can set the value of the *sqlstate* argument to return a data exception:

```
strcpy(sqlstate, "U0005");
```

In a BTEQ session, the exception condition appears in the following format, where *dbname* is the name of the database of the function:

```
*** Failure 7504 in UDF/XSP  dbname.STD_DEV: SQLSTATE U0005:
```

## Related Information

FOR more information on ...	SEE ...
using a warning condition to debug a function	<a href="#">Forcing an SQL Warning Condition.</a>

## Using Standard C Library Functions

A UDF can call standard C library functions that do not do any I/O, such as string library functions.

## malloc and free

If a UDF needs to allocate memory for its usage, you should use Teradata C library functions `FNC_malloc` and `FNC_free` instead of the standard C library functions `malloc` and `free`. The `sqltypes_td.h` header file redefines `malloc` and `free` to call `FNC_malloc` and `FNC_free`.

`FNC_free` and `FNC_malloc` check to make sure the UDF releases all memory before exiting and give an exception on the transaction if the UDF does not release all temporary memory it allocated. This prevents memory leaks in the database.

If you circumvent the logic to call `malloc` and `free` directly, there is a good chance that a memory leak could occur even if the UDF frees up memory correctly. The reason is that a UDF can abort while it is running if a user aborts the transaction, or if the transaction aborts because of some constraint violation that might occur on another node unbeknownst to the running UDF.

WHEN you ...	THEN ...
develop, test, and debug a UDF standalone	include the <code>malloc.h</code> header file and use the standard <code>malloc</code> and <code>free</code> C library functions.
use <code>CREATE FUNCTION</code> to submit the UDF source code to the server	do not include the <code>malloc.h</code> header file and use the definitions of <code>malloc</code> and <code>free</code> from the <code>sqltypes_td.h</code> header file. The definitions are used when submitting UDF source code to the server, but not when submitting UDF objects.

## C++ new and delete Operators

For C++ UDFs and external procedures, you can use the C++ `new` or `delete` operators. Teradata overwrites them by using `FNC_malloc` and `FNC_free`. The UDF or external procedure must catch the `std::bad_alloc` exception in the case where Teradata cannot fulfill a memory allocation request. Note that some `new` operator calls may be implicitly hidden in a C++ library, so you must ensure that the UDF or external procedure will catch exceptions from those library functions also.

## Other Operating System Functions

Do not use other memory allocation functions such as `calloc`, `cfree`, `clalloc`, `mlalloc`, `realloc`, or `relalloc`.

Other calls to the operating system are not allowed because unknown results can occur and possibly crash the database, if not immediately, then possibly in the future.

## Problems Using System V IPC and POSIX IPC

Teradata recommends that UDFs, UDMs, and external stored procedures not use System V IPC or POSIX IPC such as semaphore, mutex, conditional variable, and shared memory. The Teradata C library does not provide FNC functions to allocate and deallocate these IPC resources. When a session aborts or unexpected events cause Teradata to terminate the external routine execution threads or processes, there

is a risk that these IPC resources may be left over in the system without being cleaned up. In rare cases, this may cause a system hang or OS resource leak.

## Installing the Function

After you write and test a function, you can install it on the server.

### Note:

In general, you should not create UDFs, UDMs, or external stored procedures in Teradata system databases such as SYSLIB or SYSUDTLIB. These databases are primarily used for Teradata system UDFs, UDTs, UDMs, and external stored procedures only, and they usually contain a large number of these system external routines. Every time you create, alter, or drop your external routine in these databases, Teradata must relink your routine to all the objects of the system external routines. In addition, to execute your routine, Teradata must load all the shared libraries referenced by the system external routines, and these libraries may not be related to your routine. This is very inefficient. However, note that there are cases where you have to create your UDF in a system database. For example, UDFs used for row level security must reside in the SYSLIB database.

## CREATE FUNCTION Statement

Use the CREATE FUNCTION statement to identify the file name and location of the source code and install it on the server.

The function is compiled, linked to the dynamic linked library (DLL or SO) associated with the database in which the function resides, and distributed to all database nodes in the system.

Note that all the UDFs and external stored procedures that are defined in a specific database are linked into a single dynamically linked library.

## Default and Temporary Paths

To manage UDFs, Teradata uses default and temporary paths for UDF creation and execution, including:

- A default directory that Teradata uses to search for UDF source files and object files if the CREATE FUNCTION does not explicitly specify the location.

If a source file or object file is on the server and the path is not fully specified in the CREATE FUNCTION statement, the full path of the file is expected to begin here.

An administrator, or someone with sufficient privileges, is responsible for creating this directory.

- A temporary directory where UDFs are compiled.

Any files needed for the compilation process are moved here. This includes source files from the server or client as well as object and header files, if needed.

- A directory where Teradata saves the dynamically linked libraries.
- A directory of shared memory files for UDFs that execute in protected mode.

For information, including the names of UDF default and temporary paths, see the Cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## Specifying Source File Locations

The CREATE FUNCTION statement provides clauses that specify the name and path of the function source code.

CREATE FUNCTION Clause	Description
EXTERNAL	<p>Use the EXTERNAL clause when the function source is in the current or default directory on the client, and no other files need to be included.</p> <p>If the CREATE FUNCTION includes the SPECIFIC clause, then the source name is the same as the specific name; otherwise, the source name is the name that immediately follows the CREATE FUNCTION keywords.</p> <p>If the client is...</p> <ul style="list-style-type: none"> <li>workstation-attached, then BTEQ adds appropriate file extensions to the source name to locate the source file.</li> <li>mainframe-attached, then the source name must be a DDNAME file name.</li> </ul>
EXTERNAL NAME <i>function_name</i>	<p>Use the EXTERNAL <i>function_name</i> clause when the function source is in the current or default directory on the client, and no other files need to be included.</p> <p>The source name is the same as <i>function_name</i>.</p> <p>If the client is...</p> <ul style="list-style-type: none"> <li>workstation-attached, then BTEQ adds appropriate file extensions to <i>function_name</i> to locate the source file.</li> <li>mainframe-attached, then <i>function_name</i> must be a DDNAME file name.</li> </ul>
EXTERNAL NAME ' <i>string</i> '	<p>Use '<i>string</i>' to specify names and locations of the following:</p> <ul style="list-style-type: none"> <li>Function source, include header files, object files, libraries, and packages on the server.</li> <li>Function source, include header files, and object files on the client.</li> </ul> <p>You can also use '<i>string</i>' to specify that the source or include files not be stored.</p> <p>If the client is...</p> <ul style="list-style-type: none"> <li>workstation-attached, then if necessary, BTEQ adds appropriate file extensions to the names to locate the files.</li> <li>mainframe-attached, then the names must be DDNAME file names.</li> </ul>

## Source File Locations for ODBC

If you use ODBC, you can only create UDFs from files that are stored on the server.

## Source File Locations for JDBC

Using Teradata Driver for the JDBC Interface, you can create UDFs from files that are located on the Teradata server, or from resources located on the client.

A client-side UDF source file must be available as a resource in the class path. The Teradata JDBC driver can load the resource from the class path and transfer it to the server node without directly accessing the client file system.

## Specifying the C/C++ Function Name

The CREATE FUNCTION statement provides clauses that identify the C/C++ function name that appears immediately before the left parenthesis in the C/C++ function declaration or the function entry name when the C/C++ object is provided instead of the C/C++ source.

IF CREATE FUNCTION specifies this clause ...	THEN ...
EXTERNAL	<p>If the SPECIFIC clause is...</p> <ul style="list-style-type: none"> <li>specified, then the C/C++ function name must match the name that follows the SPECIFIC clause.</li> <li>not specified, then the C/C++ function name must match the name that follows the CREATE FUNCTION keywords.</li> </ul> <p>If the client is mainframe-attached, then the C/C++ function name must be the DDNAME for the source.</p>
EXTERNAL NAME <i>function_name</i>	<p>the C/C++ function name must match <i>function_name</i>.</p> <p>If the client is mainframe-attached, then <i>function_name</i> must be the DDNAME for the source.</p>
EXTERNAL NAME <i>'string'</i>	<p><i>'string'</i> can include the F option to specify the C/C++ function name.</p> <p>If <i>'string'</i> does not include the F option, the following rules apply:</p> <p>If the SPECIFIC clause is...</p> <ul style="list-style-type: none"> <li>specified, then the C/C++ function name must match the name that follows the SPECIFIC clause.</li> <li>not specified, then the C/C++ function name must match the name that follows the CREATE FUNCTION keywords.</li> </ul>

## Specifying User-Supplied Include Files

If a UDF includes a user-supplied header file, the EXTERNAL clause in the CREATE FUNCTION statement must specify the name and path of the header file.

Consider the following function that includes the header file stypes.h:

```

/***** C source file name: substr.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include "stypes.h"

```

```
void udf_substr( VARCHAR_LATIN *inputString,
                INTEGER          *start,
                VARCHAR_LATIN *result,
                char              sqlstate[6])
{
    ...
}
```

Here is an example of CREATE FUNCTION that specifies the name and path of the user-supplied header file:

```
CREATE FUNCTION udfSubStr
  (inputString VARCHAR(512),
   start      INTEGER)
RETURNS VARCHAR(512)
LANGUAGE C
NO SQL
EXTERNAL NAME 'CI!stypes!udfsrc/stypes.h!CS!substr!udfsrc/substr.c!F!udf_substr'
PARAMETER STYLE TD_GENERAL;
```

This part of the string that follows EXTERNAL NAME ... Specifies ...	
!	a delimiter.
C	that the header file is obtained from the client.
I	that the information between the following two sets of delimiters identifies the name and location of an include file (.h).
stypes	the name, without the file extension, of the header file.
udfsrc/stypes.h	the path and name of the header file on the client.
C	that the source is obtained from the client.
S	that the information between the following two sets of delimiters identifies the name and location of a C or C++ function source file.
substr	the name, without the file extension, that the server uses to compile the source.
udfsrc/substr.c	the path and name of the source file.
F	that the information after the next delimiter identifies the C or C++ function name.
udf_substr	the C or C++ function name.

For more information on installing libraries, see the information about CREATE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Troubleshooting "Arg list too long" and "Argument list too long" Errors

On Linux, creating, replacing, or altering a UDF, UDM, or external stored procedure (collectively called *external routines*) can produce an "Argument list too long" error.

If you get this error, the database in which you want to create, replace, or alter the external routine either already contains a lot of other external routines or contains external routines with very long names.

The new external routine compiles (or recompiles) successfully, but when the name of the resulting object file is added to the list of object files for all of the external routines already in the database, the compiler cannot create a new dynamic linked library (.so file) because the resulting list exceeds the maximum command line argument size.

To work around this limitation on Linux, you can do any of the following:

- Drop all unnecessary UDTs and external routines from the database.
- Instead of installing each external routine individually, create a package or library of external routines and install them collectively.

---

### Note:

This may not work in all cases, for example, if you reach the maximum number of UDTs allowed on a system.

---

## Related Information

FOR more information on ...	SEE the following documents ...
the CREATE FUNCTION statement	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the privileges that apply to UDFs	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Debugging a User-Defined Function

When debugging a function, be sure your tests include the following:

- Limit checks on the input values
- Limit checks on the return value
- Proper handling of NULLs
- Division by zero
- All memory acquired by *malloc* is freed



- All open operating system handles are released/closed

The Teradata C/C++ UDF Debugger allows some user-defined functions to be debugged within the database on a development or test system. The debugger only supports C and C++ functions. For more information on the Teradata C/C++ UDF Debugger, see [C/C++ Command-line Debugging for UDFs](#).

For other user-defined functions, debugging is limited after the function is installed in the database. You can use other debugging tools to verify their functionality, or run them on a stand-alone virtual machine and debug the UDF server processes that executes them.

For an example of how to run, test and debug your function outside of Vantage, see the Teradata Downloads article, [Developing Database Extensions \(UDFs, etc.\)](#).

## Forcing an SQL Warning Condition

You can force an SQL warning condition at the point in the code where the function appears to have problems. To force the warning, set the *sqlstate* return argument to '01H xx', where you choose the numeric value of xx.

If you use parameter style SQL, you can also set the *error\_message* return argument to return up to 256 SQL\_TEXT characters.

The warning is issued as a return state of the function. The warning does not terminate the transaction; therefore, you must set the return value of the function to a valid value that the transaction can use.

Only one warning can be returned per request.

Here is an example of how to set the SQLSTATE result code and error message return argument in a scalar function that uses parameter style SQL:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void String_UDF(VARCHAR_LATIN *inputString,
               CHARACTER_LATIN *result,
               int *inputStringIsNull,
               int *resultIsNull,
               char sqlstate[6],
               SQL_TEXT extname[129],
               SQL_TEXT specific_name[129],
               SQL_TEXT error_message[257])
{
    if (strlen((char *)inputString) == 0) {
        *result = 'F';

        /* Force a warning using the SQLSTATE result code value */
        strcpy(sqlstate, "01H01");
    }
}
```

```

    /* Set the error message return value */
    strcpy((char *)error_message, "Zero length input string");
    return;
}

...

}

```

In a BTEQ session, if String\_UDF is passed a zero length string, a warning is reported with the SQLSTATE result code value and the error message return value:

```

SELECT String_UDF('');

*** Query completed. One row found. One column returned.
*** Warning: 7505 in UDF dbname.String_UDF: SQLSTATE 01H01:
    Zero length input string
*** Total elapsed time was 1 second.

```

## Debugging Using Trace Tables

### Overall Procedure

Here is a synopsis of the steps you take to debug scalar, aggregate, and table functions using trace tables.

---

#### Note:

You can also use this procedure to debug external stored procedures.

---

1. Create a trace table using the CREATE GLOBAL TEMPORARY TRACE TABLE statement.  
The first two columns of the trace table are used by the Teradata function trace subsystem. Any columns that are defined beyond the first two are available for a UDF to use to write trace output during execution.
2. Enable the trace table for function trace output using the SET SESSION FUNCTION TRACE statement.  
You can specify an optional function trace string that the UDF can obtain during execution.
3. Invoke the function.
4. Call FNC\_Trace\_String in the UDF to get the function trace string that was specified in the SET SESSION FUNCTION TRACE statement.

You can use the value of the trace string to determine what to output to the trace table.

5. Call `FNC_Trace_Write_DL` to write trace output to the columns of a trace table.

-OR-

If you are debugging an aggregate or table function, you can optionally call `FNC_Trace_Write`, which is restricted to UDFs that execute on an AMP.

6. Use a `SELECT` statement to query the trace table and retrieve the trace output from the UDF.

## Obtaining a Trace String

To get the function trace string that was specified in the `SET SESSION FUNCTION TRACE` statement, call `FNC_Trace_String` in the UDF. You can use the value of the trace string to determine what to output to the trace table.

The following statements call `FNC_Trace_String`, which passes back the trace string in the *trace\_string* argument:

```
SQL_TEXT trace_string[257];

FNC_Trace_String(trace_string);
```

## Writing Trace Output to a Trace Table

When you create a trace table with a `CREATE GLOBAL TEMPORARY TRACE TABLE` statement, you define columns that a UDF can use to write trace output during execution.

To write trace output to the trace table columns, call `FNC_Trace_Write_DL` in the UDF. The call takes three arguments:

- An array of pointers to data to store in the trace table columns
- A count of the array elements
- An array of the length of each array element

Consider the following trace table definition:

```
CREATE GLOBAL TEMPORARY TRACE TABLE UDF_Trace
  (vproc_ID BYTE(2)
  ,Sequence INTEGER
  ,UDF_name CHAR(15)
  ,x_value INTEGER)
ON COMMIT DELETE ROWS;
```

The Teradata function trace subsystem writes values to the first two columns in the trace table.

Column	Contents
1	PE or AMP vproc number on which the UDF is running
2	<p>If FNC_Trace_Write_DL is called from ...</p> <ul style="list-style-type: none"> <li>a UDF running on a PE, then the value in the second column is a sequence number that increments by one for any call to FNC_Trace_Write_DL until the session logs off, regardless of which function makes the call.</li> <li>a UDF running on an AMP, then the value in the second column is one more than the last sequential number written to the AMP for the trace table.</li> </ul>

The following statements in a UDF output values to the UDF\_name and x\_value columns of the trace table:

```

INTEGER    x_value;
void        *column_list[2];
int         length[2];

column_list[0] = sfncname;          /* UDF input argument */
column_list[1] = &x_value;

length[0] = strlen((const char *)sfncname);
length[1] = sizeof(INTEGER);
x_value = 45;

FNC_Trace_Write_DL(2, column_list, length);

```

## Example: Debugging a UDF Using a Trace Table

The following statement creates a trace table that defines one column, *Trace\_Output*, for a UDF to write trace output to:

```

CREATE GLOBAL TEMPORARY TRACE TABLE UDF_Trace
  (vproc_ID BYTE(2)
  ,Sequence INTEGER
  ,Trace_Output VARCHAR(256))
ON COMMIT DELETE ROWS;

```

The following code uses the value of the trace string to determine the level of debugging:

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

```

```

void Find_Text ( VARCHAR_LATIN  *searched_string,
                 VARCHAR_LATIN  *pattern,
                 CHARACTER_LATIN *result,
                 int             *indicator_searched_string,
                 int             *indicator_pattern,
                 int             *indicator_result,
                 char            sqlstate[6],
                 SQL_TEXT       fncname[129],
                 SQL_TEXT       sfncname[129],
                 SQL_TEXT       error_message[257] )
{
    SQL_TEXT trace_string[257];
    char      trace_output[257];
    void      *argv[1];
    int       length[1];
    char      debug_level = '0';

    /* Get the trace string specified by SET SESSION FUNCTION */
    /* TRACE and use it to determine debug level.             */
    FNC_Trace_String(trace_string);
    if (trace_string[0] != 0)
        debug_level = trace_string[0];

    ...

    switch (debug_level)
    {
        case '1':
            /* Debug Level 1: Output the function name */
            sprintf(trace_output, "Function: %s", sfncname);
            break;
        case '2':
            /* Debug Level 1: Output all values */
            sprintf(trace_output,
                    "Function: %s, string: %s, pattern: %s",
                    sfncname, searched_string, pattern);
            break;
    }

    ...

    /* Output the trace string to the trace table. */
    argv[0] = trace_output;

```

```
length[0] = strlen(trace_output);
FNC_Trace_Write_DL(1, argv, length);

...

}
```

The following statement enables trace output for table UDF\_Trace and sets the trace string to 2 so that the UDF outputs all values to the trace table:

```
SET SESSION FUNCTION TRACE USING '2' FOR TABLE UDF_Trace;
```

The SET SESSION FUNCTION TRACE statement disables any previously enabled trace tables. The following statement queries the trace table to retrieve the trace output from the UDF:

```
SELECT *
FROM UDF_Trace
ORDER BY 1, 2;
```

Related Information

FOR more information on ...	SEE ...
FNC_Trace_String	<a href="#">FNC_Trace_String</a> .
FNC_Trace_Write	<a href="#">FNC_Trace_Write</a> .
FNC_Trace_Write_DL	<a href="#">FNC_Trace_Write_DL</a> .
CREATE GLOBAL TEMPORARY TRACE TABLE	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
SET SESSION FUNCTION TRACE	

Resolving UDF Server Setup Errors

When an external routine like an UDF, UDM, or external stored procedure is called, a UDF server process is acquired from the UDF server pool to execute the external routine. If there are no UDF server processes in the pool or if all of the processes in the pool are busy, then the system tries to start a new UDF server process for the request.

The startup of the new UDF server usually takes some time, especially if the UDF server is for executing Java external routines, or if the system is very busy. If the new UDF server cannot be started within the default time limit, the query that contains the UDF, UDM, or external procedure call is aborted, and you may

receive a 7583 error indicating that the UDF server setup encountered a problem. The system log may also record a 7820 error specifying that the UDF server could not stay up long enough for initialization.

If you are experiencing these errors, you can contact Teradata Support Center personnel to adjust the time limit allowed for starting a new UDF server process. For details, see the information about the Cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## UDF Invocation

A scalar UDF can appear almost anywhere a standard SQL scalar function can appear and an aggregate UDF can appear almost anywhere a standard SQL aggregate function can appear.

A table function can only appear in the FROM clause of an SQL SELECT statement. The SELECT statement containing the table function can appear as a subquery.

## Restrictions

In addition to the restrictions that apply to standard SQL scalar and aggregate functions, the following restrictions apply to UDFs:

- Scalar UDFs cannot appear in a partitioning expression of the CREATE TABLE statement.
- Aggregate UDFs cannot appear in recursive queries.

## Required Privileges

To invoke a UDF, you must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke a UDF that takes a UDT argument or returns a UDT, you must have the UDTUSAGE privilege on the SYSUDTLIB database or on the specified UDT.

## Argument List

The arguments in the function call must appear as comma-separated expressions in the same order as the function declaration parameters.

The arguments in the function call must be compatible with the parameter declarations in the function definition of an existing function, and must fit into the compatible type without a possible loss of information. For example, a BYTEINT argument in a function call is compatible with an INTEGER parameter declaration in the function definition, and also fits into the INTEGER type without any loss of information.

To pass an argument that is not compatible with the corresponding parameter type, explicitly convert the argument to the proper type in the function call.

A NULL argument is compatible with a parameter of any data type. For more information on the behavior of NULL arguments, see [Behavior When Using NULL as a Literal Argument](#).

For information on compatible types and the precedence rules, see [Compatible Types](#).

## Invoking UDFs with TD\_ANYTYPE Result Parameters

When invoking a scalar or aggregate UDF that is defined with a TD\_ANYTYPE result parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clauses to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column. You must enclose the UDF invocation in parenthesis if you use the RETURNS or RETURNS STYLE clauses.

For detailed information, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).

## Using Non-Deterministic UDFs as Conditions on an Index

A non-deterministic UDF is a UDF that does not always return identical results for identical inputs. If you omit the DETERMINISTIC clause in the CREATE FUNCTION or REPLACE FUNCTION statement, or if you specify the NOT DETERMINISTIC clause, the UDF is considered to be non-deterministic. Because non-deterministic UDFs are evaluated for each selected row, a condition on an index column that includes a non-deterministic UDF results in an all-AMP operation.

For example, consider the following table definition:

```
CREATE TABLE t1
  (c1 INTEGER
   ,c2 VARCHAR(9))
PRIMARY INDEX ( c1 );
```

Now consider the following function definition:

```
CREATE FUNCTION UDF_RAN(LowerBound INTEGER, UpperBound INTEGER)
RETURNS INTEGER
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
NOT DETERMINISTIC
EXTERNAL;
```

The following SELECT statement includes a condition on the c1 index column that invokes the non-deterministic UDF called UDF\_RAN and results in an all-AMP operation:

```
SELECT *
FROM t1
WHERE c1 = UDF_RAN(1,12);
```



## Ordering Input Arguments to Table UDFs

Some applications need to enforce the ordering of input to table UDFs. Rather than perform any sort processing of input arguments in the application or table UDF, you can use the HASH BY and LOCAL ORDER BY clauses when you invoke the table UDF in the FROM clause of the SELECT statement. The scope of input includes derived tables, views, and WITH objects.

Consider the following table definition:

```
CREATE TABLE
  tempData(tID INTEGER, tTS TIMESTAMP, x INTEGER, y INTEGER);
```

Suppose the data in the table looks something like this:

```
INSERT INTO tempData
  VALUES (1001, TIMESTAMP '2008-02-03 14:33:15', 10, 11);
INSERT INTO tempData
  VALUES (1001, TIMESTAMP '2008-02-03 14:44:20', 20, 24);
INSERT INTO tempData
  VALUES (1001, TIMESTAMP '2008-02-03 14:59:08', 31, 30);
INSERT INTO tempData
  VALUES (1002, TIMESTAMP '2008-02-04 11:02:19', 10, 11);
INSERT INTO tempData
  VALUES (1002, TIMESTAMP '2008-02-04 11:33:04', 22, 18);
INSERT INTO tempData
  VALUES (1002, TIMESTAMP '2008-02-04 11:48:27', 29, 29);
```

Now consider a table UDF called CharFromRows that produces a text string that represents all of the timestamp, x, and y values in rows that have the same value for the tID column. Furthermore, the values in the text string are ordered by timestamp. Here is the definition of the table UDF:

```
CREATE FUNCTION CharFromRows(tID INTEGER,
                             tTS TIMESTAMP,
                             x INTEGER,
                             y INTEGER)
  RETURNS TABLE(outID INTEGER, outCHAR VARCHAR(64000))
  LANGUAGE C
  NO SQL
  EXTERNAL NAME 'CS!charfromrows!udfsrc/charfromrows.c'
  PARAMETER STYLE SQL;
```

Here is a query that invokes the CharFromRows table UDF, using a nonrecursive WITH clause to hash the input by tID and value order the input on each AMP by tTS:

```
WITH wq (tID1, tTS1, x1, y1) AS
  (SELECT tID, tTS, x, y FROM tempData)
SELECT *
FROM TABLE (CharFromRows(wq.tID1, wq.tTS1, wq.x1, wq.y1)
  HASH BY tID1 LOCAL ORDER BY tTS1) tudf;
```

The output looks like this:

```
outID outCHAR
-----
1001  2008-02-03 14:33:1510112008-02-03 14:44:2020242008-02-03 14:59:083130
1002  2008-02-04 11:02:1910112008-02-04 11:33:0422182008-02-04 11:48:272929
```

For details on the HASH BY and LOCAL ORDER BY clauses, see the information about FROM TABLE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## UDF Locations

When a function call is qualified by a database name, Vantage looks for the UDF in the specified database only.

If you omit the database name, Vantage searches for the UDF in the following order:

1. The path specified by the SET SESSION UDFSEARCHPATH statement, if set. See *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
2. If the UDF implements cast, ordering, or transform functionality for a UDT, search the SYSUDTLIB database.

IF such a function ...	THEN ...
exists, and the arguments in the function call are compatible with the function parameters and follow the order of precedence	the search stops. If several functions exist, Vantage must decide which function is the best fit. For details, see <a href="#">Calling a Function That is Overloaded</a> .
does not exist, or the arguments in the function call are not compatible with the function parameters	the statement returns an error.

3. Search the default database for a function with the same name and number of parameters as the function call.

IF such a function ...	THEN ...
exists, and the arguments in the function call are compatible with the function parameters and follow the order of precedence	the search stops. If the database has several functions with the same name, Vantage must decide which function

IF such a function ...	THEN ...
	is the best fit. For details, see <a href="#">Calling a Function That is Overloaded</a> .
does not exist, or the arguments in the function call are not compatible with the function parameters	the search continues.

4. Search the SYSLIB database for a function with the same name and number of parameters as the function call.

IF such a function ...	THEN ...
exists, and the arguments in the function call are compatible with the function parameters and follow the order of precedence	the search stops. If several functions exist, Vantage must decide which function is the best fit. For details, see <a href="#">Calling a Function That is Overloaded</a> .
does not exist, or the arguments in the function call are not compatible with the function parameters	the statement returns an error.

For the rules of compatibility precedence, see [Compatible Types](#).

## Overloaded Function Invocation

### Calling a Function That is Overloaded

If several functions have the same name, then Vantage uses the following steps to determine which function to invoke.

IF...	THEN...
no argument in the function call is the NULL keyword	<p>if one of the functions has parameter types that are identical to the corresponding argument types of the function call, then Vantage selects that function.</p> <p>If no function has parameter types that are identical to the corresponding argument types of the function call, then Vantage repeats the following test for each argument of the function call, starting with the first (left):</p> <ul style="list-style-type: none"> <li>• If the argument type of the function call fits the corresponding parameter type of any of the functions, in order of compatibility precedence, then keep those functions and eliminate the others.</li> <li>• If the argument type of the function call does not fit the corresponding parameter type of any of the functions, in order of compatibility precedence, then return an error.</li> </ul> <p>For more information on the rules of compatible precedence, see <a href="#">Compatible Types</a>.</p>
any argument in the function call is the NULL keyword	<p>Vantage repeats the following test for each argument of the function call, starting with the first (left):</p> <ul style="list-style-type: none"> <li>• If the argument type of the function call fits the corresponding parameter type of any of the functions, in order of compatibility precedence, then keep those functions and eliminate the others.</li> </ul>

IF...	THEN...
	<ul style="list-style-type: none"> <li>• If the argument is the NULL keyword, then treat the NULL as a wildcard that matches any data type and continue to the next argument. Do not eliminate any function.</li> <li>• If the argument type of the function call does not fit the corresponding parameter type of any of the functions, in order of compatibility precedence, return an error.</li> </ul> <p>After testing each argument and not returning an error, Vantage uses the following rules:</p> <ul style="list-style-type: none"> <li>• If one function remains, then Vantage selects that function.</li> <li>• If more than one function remains, then the system returns an error.</li> </ul> <p>To avoid this error, you can explicitly cast the keyword NULL to the data type of the corresponding parameter of the overloaded UDF you want to use.</p> <p>For more information on the rules of compatible precedence, see <a href="#">Compatible Types</a>.</p>

## Calling Overloaded Functions with Period Type Arguments

When you call an overloaded function that has a PERIOD(TIME(*n*)), PERIOD(TIME(*n*) WITH TIME ZONE), PERIOD(TIMESTAMP(*n*)), or PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) parameter type (where *n* represents precision), and you pass a period argument specifying a particular precision, Vantage uses these steps to determine which function to invoke.

1. If one of the functions has parameter types and precision that are identical to the corresponding argument types and precision, then select that function.
2. If an implicit cast of the argument (source) to parameter (target) type of the function is possible and more than one function qualifies, then select the function in order of precedence listed below. For period data types, implicit cast is supported for these cases:

Source	Target TD 15.10 - CMS conversion
CHAR or VARCHAR	PERIOD(DATE)
CHAR or VARCHAR	PERIOD(TIME[( <i>n</i> )])
CHAR or VARCHAR	PERIOD(TIME[( <i>n</i> )] WITH TIME ZONE)
CHAR or VARCHAR	PERIOD(TIMESTAMP[( <i>n</i> )])
CHAR or VARCHAR	PERIOD(TIMESTAMP[( <i>n</i> )] WITH TIME ZONE)
PERIOD(DATE)	CHAR or VARCHAR
PERIOD(TIME[( <i>n</i> )])	CHAR or VARCHAR
PERIOD(TIME[( <i>n</i> )] WITH TIME ZONE)	CHAR or VARCHAR
PERIOD(TIMESTAMP[( <i>n</i> )])	CHAR or VARCHAR
PERIOD(TIMESTAMP[( <i>n</i> )] WITH TIME ZONE)	CHAR or VARCHAR
PERIOD(TIMESTAMP[( <i>n</i> )])	PERIOD(TIMESTAMP[( <i>n</i> )])
PERIOD(TIMESTAMP[( <i>n</i> )])	PERIOD(TIMESTAMP[( <i>n</i> )] WITH TIME ZONE)

Source	Target TD 15.10 - CMS conversion
PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)	PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)
PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)	PERIOD(TIMESTAMP[(n)])
PERIOD(TIME[(n)])	PERIOD(TIME[(n)])
PERIOD(TIME[(n)])	PERIOD(TIME[(n)] WITH TIME ZONE)
PERIOD(TIME[(n)] WITH TIME ZONE)	PERIOD(TIME[(n)] WITH TIME ZONE)
PERIOD(TIME[(n)] WITH TIME ZONE)	PERIOD(TIME[(n)])

If both Source and Target are period data type, then based on the implicit conversion allowed between period types, the order of precedence from high to low is:

Source	Target
PERIOD(TIME[(n)])	PERIOD(TIME[(n)]) PERIOD(TIME[(n)] WITH TIME ZONE)
PERIOD(TIME[(n)] WITH TIME ZONE)	PERIOD(TIME[(n)] WITH TIME ZONE) PERIOD(TIME[(n)])
PERIOD(TIMESTAMP[(n)])	PERIOD(TIMESTAMP[(n)]) PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)
PERIOD(TIMESTAMP[(n)] WITH TIME ZONE)	PERIOD(TIMESTAMP[(n)] WITH TIME ZONE) PERIOD(TIMESTAMP[(n)])

If the source is CHAR or VARCHAR and the target is a period type, the order of precedence from high to low is:

Source	Target
CHAR or VARCHAR	PERIOD(TIMESTAMP[(n)] WITH TIME ZONE) PERIOD(TIMESTAMP[(n)]) PERIOD(DATE) PERIOD(TIME[(n)] WITH TIME ZONE) PERIOD(TIME[(n)])

If the source is a period data type and the target is CHAR or VARCHAR, the order of precedence from high to low is:

Source	Target
Any period data type	CHAR VARCHAR

**Note:**

If you want to override these precedence rules, you must explicitly CAST the argument to the required parameter type.

## Example: Function Selection for Numeric Types

Consider the following Sales functions and the corresponding specific name as specified in the SPECIFIC clause of the CREATE FUNCTION statement.

Function Name and Parameters	Specific Name
Sales(Quantity INTEGER, Profit INTEGER)	S1
Sales(Quantity INTEGER, Profit FLOAT)	S2
Sales(Quantity FLOAT, Profit INTEGER)	S3
Sales(Quantity FLOAT, Profit FLOAT)	S4

The following table identifies which function Vantage invokes when none of the functions have parameter types that are identical to the corresponding argument types of the function call.

IF the data type of the first argument is ...	AND the data type of the second argument is ...	THEN Vantage invokes the function with the specific name ...
BYTEINT	INTEGER	S1. After testing the first argument, S1 and S2 remain on the list. After testing the second argument, S1 is left.
DECIMAL	SMALLINT	S3. After testing the first argument, S3 and S4 remain on the list. After testing the second argument, S3 remains on the list.  <b>Note:</b> The DECIMAL is converted to a FLOAT and the SMALLINT is converted to an INTEGER before the UDF is called.

## Example: Function Selection for Character Types

Consider the following Find\_Text functions and the corresponding specific name as specified in the SPECIFIC clause of CREATE FUNCTION.

Function Name and Parameters	Specific Name
Find_Text(SearchedString CHAR(40), Pattern VARCHAR(10))	F1
Find_Text(SearchedString VARCHAR(10), Pattern CHAR(20))	F2

Function Name and Parameters	Specific Name
Find_Text(SearchedString CHAR(20), Pattern CHAR(10))	F3

The following table identifies which function Vantage invokes when none of the functions have parameter types that are identical to the corresponding argument types of the function call.

IF the data type of the first argument is ...	AND the data type of the second argument is ...	THEN Vantage ...
VARCHAR(40)	CHAR(50)	invokes the function with the specific name F2. After testing the first argument, F2 remains on the list. After testing the second argument, F2 is left (exact match). The length of character string is not considered, so truncation is possible.
VARCHAR(80)	VARCHAR(20)	returns an error. After testing the first argument, F2 remains on the list. After testing the second argument, nothing remains on the list. F2 is not selected because the function is asking for a VARCHAR and the only option is a CHAR, which is of a lower precedence than the VARCHAR. Do not use CHAR as a parameter type if you want a function to be found with a VARCHAR argument.

## Example: Function Selection for NULL as a Literal Argument

Consider the following Extrapolate functions and the corresponding specific name as specified in the SPECIFIC clause of CREATE FUNCTION.

Function Name and Parameters	Specific Name
Extrapolate(a INTEGER, b INTEGER, c INTEGER)	S1
Extrapolate(a VARCHAR(20), b VARCHAR(20), c VARCHAR(20))	S2

The results of calling the function with NULL as a literal argument appears in the following table.

IF the function invocation is ...	THEN Vantage ...
SELECT Extrapolate(3, NULL, 9);	invokes the function name with the specific name S1.
SELECT Extrapolate('Ver', NULL, '.*.*');	invokes the function name with the specific name S2.
SELECT Extrapolate(NULL, NULL, NULL);	returns an error.

IF the function invocation is ...	THEN Vantage ...
SELECT Extrapolate(CAST(NULL AS INTEGER), NULL, NULL);	invokes the function name with the specific name S1.

## Protected Mode Function Execution

### Protected Mode Execution Option

The ALTER FUNCTION statement provides an option that controls whether Vantage invokes the function directly or runs the function indirectly as a separate process.

The option applies to functions that were created without specifying the EXTERNAL SECURITY clause in the CREATE FUNCTION or REPLACE FUNCTION statement. Functions that specify the EXTERNAL SECURITY clause are executed using separate secure server processes.

IF ALTER FUNCTION specifies ...	THEN Vantage ...
EXECUTE PROTECTED	runs the function indirectly as a separate process. If the function fails during execution, the transaction fails.
EXECUTE NOT PROTECTED	invokes the function directly. If you subsequently use REPLACE FUNCTION to replace the function, the execution mode reverts back to protected mode.

### NOTICE

If the ALTER FUNCTION statement specifies EXECUTE NOT PROTECTED, and the function fails during execution, the database software will probably restart.

Only an administrator, or someone with sufficient privileges, can use the ALTER FUNCTION statement.

### Choosing the Correct Execution Option

Use the following table to choose the correct execution option for a UDF.

IF ...	THEN use ...
you are in the UDF development phase and are debugging a function	EXECUTE PROTECTED.
the function opens a file or uses another operating system resource that requires tracking by the operating system	EXECUTE PROTECTED. Running such a function in nonprotected mode could interfere with proper Vantage operation.



IF ...	THEN use ...
the function is a computational function that does not use any operating system resources	EXECUTE NOT PROTECTED. Running a UDF in nonprotected mode speeds up the processing of the function considerably. Use this option only after thoroughly debugging the function and making sure it produces the correct output.

## Related Information

FOR more information on ...	SEE ...
protected mode process and server administration	<a href="#">Protected Mode Process and Server Administration for C/C++ External Routines.</a>
scripts to find unprotected mode UDFs and change the execution mode	<a href="#">Finding Unprotected Mode UDFs and Changing the Execution Mode.</a>
ALTER FUNCTION and the EXECUTE PROTECTED option	<i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.
CREATE FUNCTION and the 'D' character in the EXTERNAL NAME 'string' clause	
CREATE FUNCTION and the EXTERNAL SECURITY clause	
the privileges associated with UDFs	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Argument and Result Behavior

### Truncation of Character String Arguments and Result

The session transaction mode affects character string truncation.

IF the session transaction mode is ...	AND the function ...	THEN Vantage ...
Teradata	is passed a CHAR or VARCHAR input argument or returns a CHAR result that exceeds the maximum length specified in the CREATE FUNCTION or REPLACE FUNCTION statement	truncates the argument or result without reporting an error. Truncation on Kanji1 character strings containing multibyte characters might result in truncation of one byte of the multibyte character.
ANSI	is passed a CHAR or VARCHAR input argument or returns a CHAR result that exceeds the maximum length specified	truncates the argument or result of excess pad characters without reporting an error.

IF the session transaction mode is ...	AND the function ...	THEN Vantage ...
	in the CREATE FUNCTION or REPLACE FUNCTION statement	Truncation of other characters results in a truncation exception.

## VARCHAR and VARBYTE Result Behavior

Vantage does not truncate VARCHAR or VARBYTE return values.

Any attempt by the function to return a VARCHAR or VARBYTE value where the length is greater than the length specified in the RETURNS clause of the CREATE FUNCTION or REPLACE FUNCTION statement generates a 7509 error.

For details on using the C types that map to VARCHAR and VARBYTE, see [SQL Data Type Mapping](#).

## Behavior When Using NULL as a Literal Argument

The RETURNS NULL ON NULL INPUT and CALLED ON NULL INPUT options in the CREATE FUNCTION statement determine what happens if the NULL keyword is used as any of the input arguments.

IF an input argument is the NULL keyword and the corresponding CREATE FUNCTION statement specifies ...	AND the parameter style is ...	THEN ...
RETURNS NULL ON NULL INPUT	SQL or TD_GENERAL	the function is not evaluated and the result is always NULL.
CALLED ON NULL INPUT	SQL	the function is called with the appropriate indicators set to the null indication.
	TD_GENERAL	an error is reported.

### Note:

For table functions, the parameter style is always SQL.

If the CREATE FUNCTION statement does not specify either RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT, the default is CALLED ON NULL INPUT.

NULL as a literal argument is compatible with a parameter of any data type. For example, consider the following scalar function:

```
CREATE FUNCTION sales( p1 INTEGER, p2 DECIMAL(2,0), p3 VARCHAR(20) )
RETURNS INTEGER
LANGUAGE C
```

```
NO SQL
PARAMETER STYLE SQL
EXTERNAL;
```

You can successfully pass the NULL keyword as any function argument:

```
SELECT sales(3, NULL, 'UPD PRD-3011');
SELECT sales(NULL, NULL, NULL);
```

Passing the NULL keyword as an argument to overloaded functions can result in errors unless Vantage can identify which function to invoke without ambiguity. For details, see [Calling a Function That is Overloaded](#).

## Overflow and Numeric Arguments

To avoid numeric overflow conditions, the C or C++ function should define a decimal data type as big as it can handle.

If the assignment of the value of an input or output numeric argument would result in a loss of significant digits, a numeric overflow error is reported.

For example, consider a scalar function that takes a DECIMAL(2,0) argument:

```
CREATE FUNCTION UDF_SMLDEC( p1 DECIMAL(2,0) )
RETURNS DECIMAL(2,0)
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL;
```

Passing a number with a maximum of two digits is successful:

```
SELECT UDF_SMLDEC(99);
```

An attempt to pass a number larger than 99 or smaller than -99 would result in a loss of significant digits.

```
SELECT UDF_SMLDEC(100);
```

```
Failure 2616 Numeric overflow occurred during computation.
```

Any fractional numeric data that is passed or returned that does not fit as it is being assigned is rounded according to the Teradata rounding rules. For details, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Data Type Conversion of Arguments

### Compatible Data Type Conversion

If Vantage invokes a function that defines compatible parameter types, but not exact parameter types, input arguments are converted to the type expected in the function definition.

UDF arguments do not follow the implicit type conversion rules that permit the assignment and comparison of some types without requiring explicit conversion. For example, Vantage implicitly converts DATE types to numeric types for assignment.

If the arguments of the calling function are not compatible with an existing function, they must be explicitly cast to the proper type.

Vantage does not perform implicit data type conversions for TD\_ANYTYPE parameters.

### Numeric Constants

Teradata converts numeric constants as follows.

IF a UDF is passed this numeric constant value ...	THEN Teradata uses this numeric data type to determine the precedence order ...
-128 to 127	BYTEINT
-32768 to 32767	SMALLINT
-2147483648 to 2147483647	INTEGER
Values outside the INTEGER range and not more than 38 digits	DECIMAL
Values with a decimal point and not more than 38 digits	
Values with a decimal point and an exponent	REAL

### String Constants

Teradata converts string constants as follows.

IF a UDF is passed this constant value ...	THEN Teradata uses this data type to determine the precedence order ...
String in apostrophes	VARCHAR
Graphic string in apostrophes	VARCHAR CHARACTER SET GRAPHIC

Teradata does not implicitly convert character string constants that represent numeric values to numeric types unless they are part of an expression. For example, the string constant '1024' remains a character

string. However, in an expression such as '1024' - 1, Teradata does convert the character string to a REAL, according to Teradata implicit conversion rules.

For more information on implicit type conversion rules, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Hexadecimal Constants

Teradata converts hexadecimal constants as follows.

IF a UDF is passed a constant value that uses this hexadecimal notation ...	THEN Teradata uses this data type to determine the precedence order ...
'hexadecimal_digits'XC 'hexadecimal_digits'XCV	VARCHAR
'hexadecimal_digits'XCF	CHAR
'hexadecimal_digits'XBV	VARBYTE
'hexadecimal_digits'XB 'hexadecimal_digits'XBF	BYTE
'hexadecimal_digits'X 'hexadecimal_digits'XI 'hexadecimal_digits'XI4	INTEGER
'hexadecimal_digits'XI8	BIGINT
'hexadecimal_digits'XI2	SMALLINT
'hexadecimal_digits'XI1	BYTEINT

## Defining Functions that Use LOB Types

Large object (LOB) data types have the capacity to store large amounts of data. The maximum length of a Teradata LOB is approximately 2GB.

A *character large object* (CLOB) column can store character data, such as simple text, HTML, or XML documents. A *binary large object* (BLOB) column can store binary objects, such as graphics, audio and video clips, files, and documents.

You can use UDFs to extend the set of SQL functions that operate on LOBs.

## Function Parameter List

UDFs can define CLOB or BLOB arguments and return values, passing them by locator.

Here is an example of how to declare a scalar function that uses LOB types.

```

/***** C source file name: udfsamp.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void do_something( LOB_LOCATOR      *a,
                  LOB_RESULT_LOCATOR *result,
                  char               sqlstate[6] )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION DO_SOMETHING(A BLOB AS LOCATOR)
    RETURNS BLOB AS LOCATOR
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!udfsamp!td_udf/udfsamp.c';

```

For details on the LOB\_LOCATOR and LOB\_RESULT\_LOCATOR C data types, see [C Data Types](#).

## Accessing the Value of a Large Object

Teradata provides the following library functions for accessing the value of a large object:

- FNC\_LobOpen
- FNC\_LobRead
- FNC\_LobClose

To access the value of a large object, follow these steps:

1. Call FNC\_LobOpen to establish a read context for the large object.

FNC\_LobOpen returns a read context handle for the large object through an output argument of type LOB\_CONTEXT\_ID. The sqltypes\_td.h header file defines LOB\_CONTEXT\_ID like this:

```
typedef long LOB_CONTEXT_ID;
```

2. Call FNC\_LobRead to read the large object and place the data into a BYTE buffer.

One of the input arguments to FNC\_LobRead is the read context handle that was returned by FNC\_LobOpen.

3. Call FNC\_Close to release all resources associated with the read context.

For detailed information on each function, see [C Library Functions](#).

Here is a code excerpt that shows how to access the value of a large object:

```
#define BUFFER_SIZE 500

void do_something ( LOB_LOCATOR *a,
                   LOB_RESULT_LOCATOR *result,
                   char sqlstate[6] )
{
    BYTE          buffer[BUFFER_SIZE];
    FNC_LobLength_t actlen;
    LOB_CONTEXT_ID id;

    /* Establish a read context for the LOB input argument */
    FNC_LobOpen(*a, &id, 0, BUFFER_SIZE);

    /* Read the LOB and place the data into a BYTE buffer */
    FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen);

    /* Release the resources associated with the read context */
    FNC_LobClose(id);

    ...
}
```

For a complete example, see [C Scalar Function Using LOBs](#).

## Appending Data to a Large Object Result

A UDF that declares the result as a `LOB_RESULT_LOCATOR` type must use the `FNC_LobAppend` library function to append a sequence of bytes to the result.

The following code excerpt modifies the preceding code excerpt to show how to append data to a `LOB_RESULT_LOCATOR` result:

```
#define BUFFER_SIZE 500

void do_something ( LOB_LOCATOR *a,
                   LOB_RESULT_LOCATOR *result,
                   char sqlstate[6] )
{
    BYTE          buffer[BUFFER_SIZE];
    FNC_LobLength_t actlen;
    LOB_CONTEXT_ID id;
```

```
FNC_LobOpen(*a, &id, 0, BUFFER_SIZE);
FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen);
FNC_LobAppend(*result, buffer, actlen, &actlen);
FNC_LobClose(id);

...

}
```

### Passing LOB Data in the Intermediate Aggregate Storage Area

A locator is valid only during the execution of the UDF to which it is passed as an argument. Because of this limitation, a locator cannot be passed in the intermediate aggregate storage area that aggregate functions use to accumulate summary information.

Instead, an aggregate function must convert a locator to a *persistent object reference*, a reference that is valid over the execution of the SQL request.

The `sqltypes_td.h` header file defines a persistent object reference as a `LOB_REF` type:

```
typedef struct LOB_REF{
    unsigned char data[50];
} LOB_REF;
```

A `LOB_REF` is not valid outside of the request in which it is created. If you attempt to pass a `LOB_REF` out of the request, perhaps by storing it in a table, and later attempt to convert it back to a locator, the function will fail.

To convert a locator to or from a persistent object reference that can be passed in the intermediate aggregate storage area, Teradata provides the following library functions:

- `FNC_LobLoc2Ref`
- `FNC_LobRef2Loc`

FOR ...	SEE ...
detailed information on <code>FNC_LobLoc2Ref</code> and <code>FNC_LobRef2Loc</code> , including examples of how to use the functions	<a href="#">C Library Functions</a>
a complete example of an aggregate function using LOBs	<a href="#">C Aggregate Function Using LOBs</a>



## Type Conversion and Performance

As it does with other functions and operators, Vantage performs implicit conversion of types that are compatible with LOBs according to the rules of precedence. If you define a UDF that declares a LOB type parameter, you can call the UDF and pass a compatible argument.

IF the UDF declares this parameter type ...	THEN you can call the UDF and pass this argument type ...
CLOB	<ul style="list-style-type: none"> <li>• VARCHAR</li> <li>• CHAR</li> </ul>
BLOB	<ul style="list-style-type: none"> <li>• VARBYTE</li> <li>• BYTE</li> </ul>

In some cases, implicit conversions of types that are compatible with LOBs have performance consequences.

For example, suppose a UDF declares a BLOB parameter. If you call the UDF and pass in a VARBYTE column, Vantage converts the VARBYTE value into a temporary BLOB and passes it to the UDF. Because Vantage stores the temporary BLOB on disk, the performance cost of the conversion is significant.

A solution is to create an overloaded function that explicitly declares a VARBYTE argument.

## Truncation and Performance

Consider a UDF that declares a BLOB(100000) AS LOCATOR parameter. If you call the UDF and pass in a BLOB(200000) column, truncation might occur, so Vantage must create a temporary BLOB and perform a data copy. Because Vantage stores the temporary BLOB on disk, the performance cost of the conversion is significant.

A solution is to omit the size of the BLOB argument in the function declaration, and use the default size, which is 2,097,088,000 bytes, the maximum size of a LOB.

## Getting the Length of a LOB

To get the length in bytes of a LOB, use the FNC\_GetLobLength Teradata library function. The function returns the length as an FNC\_LobLength\_t type, defined in the sqltypes\_td.h header file like this:

```
typedef unsigned long FNC_LobLength_t;
```

For detailed information on FNC\_GetLobLength, see [C Library Functions](#).

## Related Information

FOR more information on ...	SEE ...
scalar UDFs that define LOB arguments and return values	the UDF code example in <a href="#">C Scalar Function Using LOBs</a> .
aggregate UDFs that define LOB arguments and return values	the UDF code example in <a href="#">C Aggregate Function Using LOBs</a> .
LOB access library function	<a href="#">C Library Functions</a> .

## Defining Functions that Use UDT Types

UDTs are custom data types that allow you to model the structure and behavior of data that your application deals with.

Teradata supports two types of UDTs: *distinct* types and *structured* types. A distinct type is based on a single predefined data type such as INTEGER or VARCHAR. A structured type consists of one or more named attributes that can be predefined types or other UDTs.

In addition to specifying distinct and structured UDTs as the data types of columns in table definitions, you can specify them as the data types of scalar, aggregate, and table UDF parameters and return values.

Teradata also supports a form of structured UDT called *dynamic* UDT. Instead of using a CREATE TYPE statement to define the UDT, like you use to define a distinct or structured type, you use the NEW VARIANT\_TYPE expression to declare an instance of a dynamic UDT and define the attributes of the UDT at run time.

You can specify a dynamic UDT as the data type of input parameters to UDFs. They cannot appear anywhere else.

## Function Parameter List

UDFs can define UDT arguments and return values, passing them by UDT handle. Here is an example of how to declare a scalar function that uses UDT types:

```

/***** C source file name: udfsamp.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void do_something( UDT_HANDLE      *aUDT,
                  UDT_HANDLE      *resultUDT,
                  char             sqlstate[6] )
{

```

```
    ...  
}
```

Suppose you create a structured UDT called circleUDT that you want to use as the input parameter and return type for the do\_something function. The corresponding CREATE FUNCTION statement looks like this:

```
CREATE FUNCTION DO_SOMETHING(A circleUDT)  
    RETURNS circleUDT  
    LANGUAGE C  
    NO SQL  
    PARAMETER STYLE TD_GENERAL  
    EXTERNAL NAME 'CS!udfsamp!td_udf/udfsamp.c';
```

To use a dynamic UDT as the input parameter, the corresponding CREATE FUNCTION statement looks like this:

```
CREATE FUNCTION DO_SOMETHING(A VARIANT_TYPE)  
    RETURNS circleUDT  
    LANGUAGE C  
    NO SQL  
    PARAMETER STYLE TD_GENERAL  
    EXTERNAL NAME 'CS!udfsamp!td_udf/udfsamp.c';
```

For details on the UDT\_HANDLE C data type, see [C Data Types](#).

## Accessing the Value of a Distinct UDT

To access the value of a distinct UDT, follow these steps.

IF the distinct type ...	THEN ...
does not represent a LOB	<ol style="list-style-type: none"><li>1. Allocate a buffer using the C data type that maps to the underlying type of the distinct UDT.</li><li>2. Call FNC_GetDistinctValue to place the value of the UDT into the buffer you allocated.</li></ol>
represents a LOB	<ol style="list-style-type: none"><li>1. Declare a LOB_LOCATOR to hold the locator of the distinct type.</li><li>2. Call FNC_GetDistinctInputLob to set the LOB_LOCATOR to the LOB locator of the distinct UDT.</li><li>3. Follow the steps in <a href="#">Accessing the Value of a Large Object</a> to read the data.</li></ol>

For detailed information on FNC\_GetDistinctValue and FNC\_GetDistinctInputLob, see [C Library Functions](#).

Here is a code excerpt that shows how to access the value of a distinct type that represents a FLOAT:

```
void meters_t_toInches( UDT_HANDLE    *metersUdt,
                       FLOAT          *result,
                       char            sqlstate[6])
{
    FLOAT value;
    int length;
    /* Get the value of metersUdt. */
    FNC_GetDistinctValue(*metersUdt, &value, sizeof_FLOAT, &length);

    ...
}
```

## Accessing the Attribute Values of a Structured UDT

To access the attribute values of structured UDTs (including dynamic UDTs), follow these steps:

1. [Optional] Call FNC\_GetStructuredAttributeCount to get the number of attributes in the UDT.
2. [Optional] Call FNC\_GetStructuredAttributeInfo to get information, such as the data type, about the attributes in the UDT.
3. Access the attribute value based on the data type of the attribute.

IF the attribute ...	THEN ...
does not represent a LOB	Allocate a buffer using the C data type that maps to the underlying type of the attribute. Call FNC_GetStructuredAttribute or FNC_GetStructuredAttributeByNdx to place the value of the attribute into the buffer you allocated.
represents a LOB	Declare a LOB_LOCATOR to hold the locator of the attribute. Call FNC_GetStructuredInputLobAttribute or FNC_GetStructuredInputLobAttributeByNdx to set the LOB_LOCATOR to the LOB locator of the attribute. Follow the steps in <a href="#">Accessing the Value of a Large Object</a> to read the data.

For details on FNC\_GetStructuredAttributeCount, FNC\_GetStructuredAttributeInfo, FNC\_GetStructuredAttribute, FNC\_GetStructuredAttributeByNdx, FNC\_GetStructuredInputLobAttribute, and FNC\_GetStructuredInputLobAttributeByNdx, see [C Library Functions](#).

Here is a code excerpt that shows how to access the value of a structured type attribute that represents an INTEGER:

```
void getX( UDT_HANDLE *pointUdt,
           INTEGER     *result,
           char        sqlstate[6])
```

```
{
    INTEGER x;
    int nullIndicator;
    int length;
    /* Get the x attribute of pointUdt. */
    FNC_GetStructuredAttribute(*pointUdt, "x", &x, sizeof_INTEGER,
                              &nullIndicator, &length);

    if (nullIndicator == -1) {
        /* do null handling here */
        ...
        return;
    }

    ...
}
```

## Setting the Value of a Distinct UDT Result

To set the value of a distinct UDT that is defined to be the result of a UDF, follow these steps.

IF the distinct type ...	THEN ...
does not represent a LOB	<ol style="list-style-type: none"><li>1. Allocate a buffer using the C data type that maps to the underlying type of the distinct UDT.</li><li>2. Place the return value into the buffer.</li><li>3. Call FNC_SetDistinctValue to set the value of the UDT to the return value in the buffer you allocated.</li></ol>
represents a LOB	<ol style="list-style-type: none"><li>1. Declare a LOB_RESULT_LOCATOR to hold the locator of the distinct type.</li><li>2. Call FNC_GetDistinctResultLob to set the LOB_RESULT_LOCATOR to the locator of the distinct UDT.</li><li>3. Follow the steps in <a href="#">Appending Data to a Large Object Result</a> to append the return data using the LOB locator.</li></ol>

For detailed information on FNC\_SetDistinctValue and FNC\_GetDistinctResultLob, see [C Library Functions](#).

The following code excerpt sets the value of a distinct UDT that represents a FLOAT:

```
void meters_t_toFeet( UDT_HANDLE    *metersUdt,
                     UDT_HANDLE    *resultFeetUdt,
                     char           sqlstate[6])
{
    FLOAT value;
    int length;
```

```

/* Get the value of metersUdt. */
FNC_GetDistinctValue(*metersUdt, &value, sizeof_FLOAT, &length);
/* Convert meters to feet and set the result value */
value *= 3.28;
FNC_SetDistinctValue(*resultFeetUdt, &value, sizeof_FLOAT);

...
}

```

## Setting the Attribute Values of a Structured UDT Result

To set the attribute values of a structured UDT that is defined to be the result of a UDF, follow these steps.

IF the attribute ...	THEN ...
does not represent a LOB	Allocate a buffer using the C data type that maps to the underlying type of the attribute. Place the value of the attribute into the buffer. Call <code>FNC_SetStructuredAttribute</code> or <code>FNC_SetStructuredAttributeByNdx</code> to set the value of the attribute to the value in the buffer you allocated. You can also set the attribute to null.
represents a LOB	Declare a <code>LOB_RESULT_LOCATOR</code> to hold the locator of the LOB attribute. Call <code>FNC_GetStructuredResultLobAttribute</code> or <code>FNC_GetStructuredResultLobAttributeByNdx</code> to set the <code>LOB_RESULT_LOCATOR</code> to the locator of the LOB attribute. Follow the steps in <a href="#">Appending Data to a Large Object Result</a> to append the return data using the LOB locator.

For details on `FNC_SetStructuredAttribute`, `FNC_SetStructuredAttributeByNdx`, `FNC_GetStructuredResultLobAttribute`, and `FNC_GetStructuredResultLobAttributeByNdx`, see [C Library Functions](#).

The following code excerpt sets the attribute value of a structured UDT:

```

void setX( UDT_HANDLE *pointUdt,
          INTEGER      *val,
          UDT_HANDLE *resultPoint,
          char          sqlstate[6])
{
    INTEGER x;
    INTEGER newval;
    int nullIndicator;
    int length;
    /* Set the x attribute of the result point. */
    nullIndicator = 0;

```

```

newval = *val;
FNC_SetStructuredAttribute(*resultPoint, "x", &newval,
                           nullIndicator, sizeof_INTEGER);

...
}

```

## Functions That Implement Functionality for a UDT

Functions that provide cast, ordering, or transform functionality for a UDT must satisfy certain requirements.

IF the UDT is ...	AND the function provides ...	THEN the C/C++ function must ...
structured	cast functionality from another UDT or predefined type to the UDT	set the attribute values of the UDT result using the value of the input argument.
	cast functionality from the UDT to another UDT or predefined type	set the value of the result using the attribute values of the UDT input argument.
	transform functionality for importing the UDT to the server	transform the value of the predefined type input argument into attribute values of the UDT result.
	transform functionality for exporting the UDT from the server	transform the attribute values of the UDT into an appropriate value for the predefined type result.
	ordering functionality for comparing two UDTs	use the attribute values of the UDT to set the result to a value that Vantage uses for comparisons.
distinct	cast functionality from another UDT or predefined type to the UDT	set the value of the UDT result using the value of the input argument.
	cast functionality from the UDT to another UDT or predefined type	set the value of the result using the value of the UDT input argument.
	transform functionality for importing the UDT to the server	transform the value of the predefined type input argument into the value of the UDT result.
	transform functionality for exporting the UDT from the server	transform the value of the UDT into an appropriate value for the predefined type result.
	ordering functionality for comparing two UDTs	use the value of the UDT to set the result to a value that Vantage uses for comparisons.

Vantage automatically generates cast functionality between a distinct type and its predefined source type. You can create additional functions for casting between a distinct type and other predefined data types or UDTs.

Although Vantage automatically generates transform and ordering functionality when you create a distinct type, you can drop the functionality and provide your own. Note that Vantage does not automatically generate ordering functionality for distinct UDTs where the source data type is a LOB.

## Functions and Geospatial Data Types

Although Teradata implements the geospatial data types ST\_Geometry and MBR as UDTs, they are neither distinct nor structured. The geospatial data types are Teradata proprietary internal UDTs (also called system defined types, or SDTs) and as such, are not useful as parameter types or the return type of an external routine.

An application that needs to pass geospatial values to an external routine can use any number of methods on an ST\_Geometry or MBR type to get the entire representation or part of the representation as a data type that can be passed to an external routine.

IF a UDF needs ...	THEN pass the ...
the entire representation of an ST_Geometry type in text format	well-known text representation of the ST_Geometry value by calling the ST_AsText method.
the entire representation of an ST_Geometry type in binary format	well-known binary representation of the ST_Geometry value by calling the ST_AsBinary method.
part of an ST_Geometry value, such as the X or Y coordinate of an ST_Point	value by calling the appropriate method, such as ST_X or ST_Y.

For more information on geospatial types, including well-known text formats, well-known binary formats, methods, functions, and stored procedures, see *Teradata Vantage™ - Geospatial Data Types*, B035-1181.

## Related Information

FOR more information on ...	SEE ...
LOB access library function	<a href="#">C Library Functions.</a>
library functions for accessing and setting the value of a distinct UDT or attribute values of a structured UDT	
a synopsis of the steps you take to develop, compile, install, and use a UDF that implements UDT transform, ordering, or cast functionality	<a href="#">UDFs that Implement UDT Functionality.</a>
code examples that show functions that use UDT parameters	<a href="#">C Aggregate Function Using UDTs.</a>
	<a href="#">C Scalar Function Using Dynamic UDTs.</a>
code examples that implement UDT functionality	<a href="#">C Scalar Functions for UDT Functionality.</a>



## Defining Functions that Use Period Types

A *period* is an anchored duration that represents a set of contiguous time granules. It has a beginning bound (defined by the value of a beginning element) and an ending bound (defined by the value of an ending element). The representation of the period extends from the beginning bound up to but not including the ending bound.

Teradata supports the following Period data types:

- PERIOD(DATE)
- PERIOD(TIME)
- PERIOD(TIMESTAMP)

PERIOD(TIME) and PERIOD(TIMESTAMP) can be specified with optional fractional seconds, for example PERIOD(TIME(6)), and optional time zone, for example PERIOD(TIMESTAMP WITH TIME ZONE).

## Function Parameter List

UDFs can define Period arguments and return values, passing them by a handle.

Here is an example of how to declare a scalar function that uses Period types:

```

/***** C source file name: udfsamp.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void do_something( PDT_HANDLE      *aPDT,
                  PDT_HANDLE      *resultPDT,
                  char             sqlstate[6] )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION DO_SOMETHING(A PERIOD(TIMESTAMP))
    RETURNS PERIOD(DATE)
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!udfsamp!td_udf/udfsamp.c';

```

For details on the PDT\_HANDLE C data type, see [C Data Types](#).

## Accessing the Value of a Period Argument

To access the value of a Period type argument, follow these steps:

1. Allocate a buffer that is large enough to hold the beginning and ending bounds of the Period type, as represented by the C data type that maps to the SQL element type.

For example, consider a CREATE FUNCTION statement that specifies a PERIOD(TIME WITH TIME ZONE) argument. To access the value of the argument, the function must allocate a buffer that is large enough to hold two ANSI\_Time\_WZone values, where ANSI\_Time\_WZone is the C data type that maps to TIME WITH TIME ZONE.

2. Call FNC\_GetInternalValue to place the value of the Period type into the buffer you allocated.

For detailed information on FNC\_GetInternalValue, see [C Library Functions](#)

Here is a code excerpt that shows how to access the value of a PERIOD(DATE) type:

```
void check_duration( PDT_HANDLE    *duration,
                    INTEGER        *result,
                    char            sqlstate[6])
{
    void * tmpBuf = 0;
    int bufSize = 0;
    int length;
    /* Get the value of the PERIOD(DATE) duration. */
    bufSize = sizeof_DATE * 2;
    tmpBuf = FNC_malloc(bufSize);
    FNC_GetInternalValue(*duration, tmpBuf, bufSize, &length);
    ...
}
```

## Setting the Value of a Period Result

To set the value of a Period return type, follow these steps:

1. Allocate a buffer that is large enough to hold the beginning and ending bounds of the Period type, as represented by the C data type that maps to the SQL element type.

For example, consider a CREATE FUNCTION statement that specifies a PERIOD(TIME WITH TIME ZONE) return type. To set the return value, the function must allocate a buffer that is large enough to hold two ANSI\_Time\_WZone values, where ANSI\_Time\_WZone is the C data type that maps to TIME WITH TIME ZONE.

2. Place the return value into the buffer.
3. Call FNC\_SetInternalValue to set the value of the Period result to the return value in the buffer you allocated.

For detailed information on FNC\_SetInternalValue, see [C Library Functions](#).

The following code excerpt sets the value of a PERIOD(DATE) result (note that the UDF is defined so that it can pass back a null for the result):

```
void set_duration( DATE      *date1,
                  DATE      *date2,
                  PDT_HANDLE *result,
                  int        *date1IsNull,
                  int        *date2IsNull,
                  int        *resultIsNull,
                  char        sqlstate[6])
                  SQL_TEXT    extname[129],
                  SQL_TEXT    specific_name[129],
                  SQL_TEXT    error_message[257] )
{
    DATE new_duration[2];

    /* Set the value of the PERIOD(DATE) result. */
    if (*date2 > *date1)
    {
        new_duration[0] = *date1;
        new_duration[1] = *date2;
    }
    else if (*date1 > *date2)
    {
        new_duration[0] = *date2;
        new_duration[1] = *date1;
    }
    else
    {
        strcpy(sqlstate, "22023");
        strcpy((char *) error_message,
            "PERIOD element values cannot be equal." );
        *resultIsNull = -1;
        return;
    }

    FNC_SetInternalValue(*result, &new_duration[0], sizeof_DATE*2);

    ...
}
```

## Related Information

FOR more information on ...	SEE ...
library functions for accessing and setting the values of Period arguments and results	<a href="#">C Library Functions.</a>
Period data types	<i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143.

## Defining Functions that Use the TD\_ANYTYPE Type

Teradata provides a parameter data type called TD\_ANYTYPE that can accept any system-defined data type or UDT. You can specify TD\_ANYTYPE as a data type for:

- Input parameters in scalar, aggregate and table functions
- Result parameters in scalar and aggregate functions
- IN, INOUT, or OUT parameters in external stored procedures
- Input parameters and return value in UDMs

You cannot use TD\_ANYTYPE as the return type in table functions.

The parameter attributes and return type are determined at execution time based on the actual arguments passed to the routine.

## Function Parameter List

A TD\_ANYTYPE parameter type can accept any system-defined data type or UDT; therefore, TD\_ANYTYPE arguments are passed in as void \*.

Here is an example of how to declare a scalar function that uses TD\_ANYTYPE types:

```

/***** C source file name: ascii.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void ascii( void * inputStr,
           void * result,
           ... )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```
CREATE FUNCTION ascii( str TD_ANYTYPE )
  RETURNS TD_ANYTYPE
  LANGUAGE C
  NO SQL
  SPECIFIC ascii
  EXTERNAL NAME 'CS!ascii!UDFs/ascii.c'
  PARAMETER STYLE SQL;
```

## Accessing Information Associated With TD\_ANYTYPE Parameters

Teradata provides the `FNC_GetAnyTypeParamInfo` library function for accessing information about each `TD_ANYTYPE` input and output parameter declared in your routine. `FNC_GetAnyTypeParamInfo` returns the following information for the parameters:

- The parameter positional index
- The data type of the parameter
- Whether the parameter is an input, output, or an INOUT parameter
- The maximum length, in bytes, for the data type
- The value  $n$  in a `DECIMAL( $n$ , $m$ )` type or the precision in Interval data types. For example:  
the value  $p$  in `INTERVAL DAY( $p$ ) TO SECOND( $n$ )`
- The value  $m$  in a `DECIMAL( $n$ , $m$ )` type or the fractional seconds precision in `TIME`, `TIMESTAMP`, and Interval data types. For example:  
the  $n$  value in `TIME( $n$ )` or `INTERVAL DAY( $p$ ) TO SECOND( $n$ )`
- The server character set associated with a character data type
- The name of the data type if the parameter is a UDT
- An indicator to show if the type is a UDT and if so, what kind of UDT
- The return type of a `TD_ANYTYPE` result parameter
- The number of `TD_ANYTYPE` parameters

For details about `FNC_GetAnyTypeParamInfo`, see [FNC\\_GetAnyTypeParamInfo](#).

## Using TD\_ANYTYPE Parameters in a Routine

Here are the basic steps to use `TD_ANYTYPE` parameters in a routine:

1. Allocate a buffer to hold information about the `TD_ANYTYPE` input and result parameters.
2. Call `FNC_GetAnyTypeParamInfo` to retrieve information about the `TD_ANYTYPE` arguments passed to the routine.
3. Execute the appropriate code according to the data type and information returned by `FNC_GetAnyTypeParamInfo` for each input parameter. Return a runtime error if a data type is not supported by the routine.

**Note:**

Vantage does not perform implicit data type conversions for TD\_ANYTYPE parameters.

4. Execute the appropriate code according to the requested data type of the return parameter. Return a runtime error if the requested return type is not supported by the routine.
5. Free the memory allocated for the buffer.

## The RETURNS and RETURNS STYLE Clauses

When invoking a scalar or aggregate UDF that is defined with a TD\_ANYTYPE result parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clauses to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

The RETURNS or RETURNS STYLE clause is not mandatory as long as the function also includes a TD\_ANYTYPE input parameter. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD\_ANYTYPE input argument is used to determine the return type of the TD\_ANYTYPE result parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.

You can use these clauses only with scalar and aggregate UDFs. You cannot use them with table functions. Also, you must enclose the UDF invocation in parenthesis if you use the RETURNS or RETURNS STYLE clauses.

## Related Information

For more information on...	See...
the TD_ANYTYPE type	<i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143.
using TD_ANYTYPE parameters to reduce the number of overloaded routines	<a href="#">Using TD_ANYTYPE Parameters as an Alternative to Overloading Function Names.</a>
the FNC_GetAnyTypeParamInfo function	<a href="#">FNC_GetAnyTypeParamInfo [Deprecated]</a> .
examples of functions that use TD_ANYTYPE parameters	<a href="#">C Scalar Function Using TD_ANYTYPE Parameters.</a> <a href="#">C Aggregate Function Using TD_ANYTYPE Parameters.</a>

## Defining Functions that Use ARRAY Types

An ARRAY data type is a named data type that has a defined maximum number of elements of the same specific data type. An ARRAY data type can be defined to be of one or more dimensions and permits many values of the same data type to be stored sequentially or in a matrix-like format. This extends the number of data values of the same type that can be stored in a table row.

Teradata supports one-dimensional (1-D) and multi-dimensional (n-D) ARRAY data types. You can define ARRAY input and output parameters for UDFs, UDMs, and external stored procedures that are written in C or C++.

## Function Parameter List

UDFs can define ARRAY arguments and return values, passing them by a handle.

Here is an example of how to declare a scalar function that uses an ARRAY type:

```

/***** C source file name: MyArrayUDF.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void MyArrayUDF( ARRAY_HANDLE    *ary_handle,
                 VARCHAR_LATIN   *result,
                 int              *indicator_ary,
                 int              *indicator_result,
                 char             sqlstate[6],
                 SQL_TEXT         extname[129],
                 SQL_TEXT         specific_name[129],
                 SQL_TEXT         error_message[257])
{
    ...
}

```

The following statement creates a 1-D ARRAY data type called `phonenumbers_ary`, which has an element type of CHAR.

```

/* Oracle-compatible syntax: */
CREATE TYPE phonenumbers_ary AS VARRAY(5) OF CHAR(10);
/* Teradata syntax: */
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];

```

If you want to use `phonenumbers_ary` as the input parameter type for the `MyArrayUDF` function, the corresponding `CREATE FUNCTION` statement looks like this:

```

CREATE FUNCTION MyArrayUDF( a1 phonenumbers_ary )
  RETURNS VARCHAR(100)
  NO SQL
  PARAMETER STYLE SQL
  DETERMINISTIC

```

```
LANGUAGE C
EXTERNAL NAME 'CS!MyArrayUDF!MyArrayUDF.c!F!MyArrayUDF';
```

For details on the `ARRAY_HANDLE` C data type, see [C Data Types](#).

## Accessing Information About ARRAY Elements

Teradata provides the following library functions to retrieve information about an `ARRAY` argument, such as the number of elements present, the element data type, and the number of dimensions in the `ARRAY`.

- `FNC_GetArrayElementCount`
- `FNC_GetArrayNumDimensions`
- `FNC_GetArrayTypeInfo`

Call `FNC_GetArrayElementCount` to get the total number of elements that are set to a value (including `NULL`), and the index to the last element that is present in the `ARRAY`.

Call `FNC_GetArrayNumDimensions` to get the number of dimensions defined for an `ARRAY` type when this information is not known to the external routine. This enables the writer of the external routine to write code so that it is dimension independent for any `ARRAY` types accessed in the routine.

`FNC_GetArrayTypeInfo` returns information about an `ARRAY` argument as follows:

- The number of dimensions defined for the `ARRAY`.
- The total number of elements (across all dimensions) defined for the `ARRAY`. This is also known as the maximum cardinality of the `ARRAY`.
- The data type of the elements. If the element type is UDT, the function also provides the UDT type and UDT type name.
- The maximum length in bytes of an element value.
- The precision and scale values for certain element types, such as `DECIMAL`.
- The server character set associated with the element if the element is a character type.
- The scope of each dimension of the `ARRAY`.

For detailed information about these functions, see [C Library Functions](#).

## Checking and Setting the NullBitVector

A `NullBitVector` identifies which elements in an `ARRAY` are present and not `NULL`, or present but is set to `NULL`.

A `NullBitVector` is an array of bytes that contain the presence bit for each initialized element in an `ARRAY`. Each byte contains 8 presence bits. The bits are stored in row-major order and are numbered from 0. A bit value of 1 means that the corresponding `ARRAY` element contains a non-null value. A bit value of 0 means that the `ARRAY` element is present but set to `NULL`.

The data type used to access the `NullBitVector` is defined in `sqltypes_td.h` as:



```
typedef unsigned char NullBitVecType;
```

Teradata provides the following library functions to check and set the bits of a NullBitVector.

Library Function	Description
FNC_CheckNullBitVector	Checks the value of one bit in a NullBitVector.
FNC_CheckNullBitVectorByElemIndex	Checks the value of one bit in a NullBitVector where the bit to be checked may be referenced by the ARRAY type element index as specified by dimension.
FNC_SetNullBitVector	Sets either one bit or all bits in a NullBitVector to the specified value.
FNC_SetNullBitVectorByElemIndex	Sets one bit in a NullBitVector where the bit to be set may be referenced by the ARRAY type element index as specified by dimension.

The following shows a typical usage of a NullBitVector:

1. Call FNC\_GetArrayTypeInfo to find out the number of elements in the ARRAY argument. Alternatively, you can call FNC\_GetArrayElementCount to find out the total number of elements that are currently present in the ARRAY argument.
2. Allocate a new NullBitVector array:

```
NullBitVector = (NullBitVecType*)FNC_malloc(nullVecBufSize);
```

where *nullVecBufSize* is based on the number of elements in the ARRAY.

3. Initialize the NullBitVector by setting all bits to 0 (not present):

```
memset(NullBitVector, 0, nullVecBufSize);
```

4. Call FNC\_GetArrayElements to get the values of one or more ARRAY elements, passing the NullBitVector to the function. FNC\_GetArrayElements modifies the NullBitVector to indicate which of the requested elements are present and not NULL, or present but set to NULL.
5. Call FNC\_CheckNullBitVector or FNC\_CheckNullBitVectorByElemIndex with the modified NullBitVector to interpret the results returned by FNC\_GetArrayElements.
6. Call FNC\_free to release allocated resources once you have processed the data.

For detailed information about these functions, see [C Library Functions](#).

## Accessing the Value of Array Elements

Teradata provides the following library functions for retrieving the element values of an ARRAY argument or to retrieve UDT handles for ARRAY arguments whose element type is UDT.

- FNC\_GetArrayElements

- `FNC_GetUDTHandles`

To access the value of one or more ARRAY elements, follow these steps:

1. Call `FNC_GetArrayTypeInfo` to find out the number of elements in the ARRAY argument. Alternatively, you can call `FNC_GetArrayElementCount` to find out the total number of elements that are currently present in the ARRAY argument.
2. Allocate and initialize a new `NullBitVector` array. For details, see [Checking and Setting the NullBitVector](#).
3. Allocate the result buffer which `FNC_GetArrayElements` will use to return the element values.
4. Allocate the structure that provides the index to the set of ARRAY elements that you want to retrieve. Set the values of the structure to the range of elements.
5. Call `FNC_GetArrayElements` to retrieve the values of the elements in the specified range, passing in the result buffer and the `NullBitVector` to the function. `FNC_GetArrayElements` returns the values of the requested elements in the result buffer. Any elements that are NULL will not have their value returned, but will instead have their presence bit set to 0 in the `NullBitVector`.
6. Call `FNC_CheckNullBitVector` or `FNC_CheckNullBitVectorByElemIndex` with the modified `NullBitVector` to interpret the results returned by `FNC_GetArrayElements`.
7. Call `FNC_free` to release allocated resources once you have processed the data.

For details, see [FNC\\_GetArrayElements](#).

If the element type of the ARRAY argument is UDT, you can call `FNC_GetUDTHandles` instead of `FNC_GetArrayElements` to access the element values. `FNC_GetUDTHandles` returns one or more UDT handles which you can pass to the appropriate UDT interface library functions to access or set the value (or attribute values) of a UDT.

For details, see [FNC\\_GetUDTHandles](#) and [UDT Interface](#).

## Setting the Value of Array Elements

Teradata provides the following library functions for setting the element values of an ARRAY result parameter.

Library Function	Description
<code>FNC_SetArrayElements</code>	Sets one or more elements of an ARRAY to the same new value.
<code>FNC_SetArrayElementsWithMultiValues</code>	Sets the value of one or more elements of an ARRAY where each element can be set to a different value.

The following procedure shows the typical steps you can use to set a range of ARRAY elements to the same new value.

1. Set the *nullIndicator* argument to 0.
2. Set the *newValue* argument to the value you want to assign to the range of elements.
3. Call `FNC_GetArrayTypeInfo` to find out the number of dimensions in the ARRAY.

4. Allocate the structure that provides the index to the set of ARRAY elements that you want to modify. Set the values of the structure to the range of elements.
5. Call `FNC_SetArrayElements` to set the value of the specified range of elements to *newValue*.
6. Call `FNC_free` to release allocated resources once you have processed the data.

For details, see [FNC\\_SetArrayElements](#).

The following procedure shows the typical steps you can use to set a range of ARRAY elements to different values.

1. Call `FNC_GetArrayTypeInfo` to find out the number of elements in the ARRAY. Alternatively, you can call `FNC_GetArrayElementCount` to find out the total number of elements that are currently present in the ARRAY argument.
2. Allocate and initialize a new `NullBitVector` array. For details, see [Checking and Setting the NullBitVector](#).
3. Allocate the structure that provides the index to the set of ARRAY elements that you want to modify. Set the values of the structure to the range of elements.
4. Allocate the *newValues* buffer to be the size corresponding to the number of elements to be modified. Fill the buffer with the desired element value for each element in the specified range, in row-major order.
5. Use `FNC_SetNullBitVector` or `FNC_SetNullBitVectorByElemIndex` to set the bits of the `NullBitVector` to indicate the elements that will be set to a value or to NULL.
6. Call `FNC_SetArrayElementsWithMultiValues` to set the specified range of elements to the new values based on the values of the *newValues* buffer and the `NullBitVector`.
7. Call `FNC_free` to release allocated resources once you have processed the data.

Alternatively, you can call `FNC_GetArrayElements` first to get multiple values from an ARRAY and a corresponding `NullBitVector`. Then you can optionally modify both of these structures and use `FNC_SetArrayElementsWithMultiValues` to update the same range of elements in a different ARRAY that has the same ARRAY data type.

For more information, see [FNC\\_SetArrayElementsWithMultiValues](#).

## Related Information

For more information on...	See...
the ARRAY data type	<i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143.
the ARRAY access library functions	<a href="#">C Library Functions</a> .

## Defining Functions for Algorithmic Compression

You can use algorithmic compression to compress table columns with the following data types:

- ARRAY
- BYTE

- VARBYTE
- BLOB
- CHARACTER
- VARCHAR
- CLOB
- JSON, with some restrictions listed below
- DATASET, with some restrictions listed below
- TIME and TIME WITH TIME ZONE
- TIMESTAMP and TIMESTAMP WITH TIME ZONE
- Period types
- Distinct UDTs, with some restrictions listed below
- System-defined UDTs, with some restrictions listed below

You can use the CREATE TABLE or ALTER TABLE statement with the COMPRESS USING and DECOMPRESS USING options to specify the names of UDFs to use to compress and decompress the data stored in the column.

For example, consider the following table definition:

```
CREATE TABLE Descriptions
  (d_ID INTEGER
   ,d_Data VARCHAR(2400) COMPRESS USING Compress_Data
                           DECOMPRESS USING Decompress_Data
  );
```

The table definition specifies that Vantage is to use a UDF named Compress\_Data to compress the character data in column d\_Data. Similarly, to decompress the compressed data for column d\_Data, Vantage is to use the UDF named Decompress\_Data.

Although Teradata supplies some functions you can use for algorithmic compression, these functions might not be suitable for your data. In such cases, you can write your own compression UDFs.

For information about the compression and decompression functions provided by Teradata, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

The following restrictions apply:

- You cannot use ALC to compress columns that have a data type of structured UDT.
- The TD\_LZ\_COMPRESS and TD\_LZ\_DECOMPRESS system functions compress all large UDTs including UDT-based system types such as Geospatial, XML, and JSON. However, if you write your own compression functions, the following restrictions apply:
  - Custom compression functions cannot be used to compress UDT-based system types (except for ARRAY and Period types).
  - Custom compression functions cannot be used to compress distinct UDTs that are based on UDT-based system types (except for ARRAY and Period types).

- You cannot write your own compression functions to perform algorithmic compression on JSON type columns. However, Teradata provides the `JSON_COMPRESS` and `JSON_DECOMPRESS` functions that you can use to perform ALC on JSON type columns.
- You cannot write your own compression functions to perform algorithmic compression on DATASET type columns. However, Teradata provides the `SNAPPY_COMPRESS` and `SNAPPY_DECOMPRESS` functions that you can use to perform ALC on DATASET type columns.
- You cannot use ALC to compress temporal columns:
  - A column defined as `SYSTEM_TIME`, `VALIDTIME`, or `TRANSACTIONTIME`.
  - The DateTime columns that define the beginning and ending bounds of a temporal derived period column (`SYSTEM_TIME`, `VALIDTIME`, or `TRANSACTIONTIME`).

You can use ALC to compress Period data types in columns that are nontemporal; however, you cannot use ALC to compress derived period columns.

For more information about temporal tables, see *Teradata Vantage™ - Temporal Table Support*, B035-1182 and *Teradata Vantage™ - ANSI Temporal Table Support*, B035-1186.

- You cannot specify multivalued or algorithmic compression for a row-level security constraint column.

UDFs that are used for algorithmic compression must be scalar UDFs.

You must thoroughly test the UDFs you develop for algorithmic compression. If the compression or decompression algorithm fails, compressed data may not be recoverable, or may be corrupted.

## Rules for Writing UDFs that Compress Character and Byte Data

- The scalar UDF must declare a single `BYTE(n)`, `VARBYTE(n)`, `CHAR(n)`, or `VARCHAR(n)` input parameter that has a data type that is compatible with the column data type. The return parameter type must be `VARBYTE(n)`. For details on compatible types, see [Compatible Types](#).

The length of the column must be less than or equal to the length of the input parameter.

For `CHAR` and `VARCHAR` columns, you must use the `CHARACTER SET` phrase in the parameter declaration to specify a server character set that matches the server character set of the column.

**Recommendation:** Declare the input parameter with a data type of `VARBYTE(n)` or `VARCHAR(n)`. This is best for most situations.

- When the input cannot be compressed, the UDF can indicate it is unable to compress the column data by setting the length field of the output `VARBYTE` argument to less than 0.
- If the column data type is `VARCHAR` or `CHAR` and you declare the UDF input parameter as `CHAR` or `VARCHAR`, then the UDF must call `FNC_GetCharLength` to get the actual length of the input string.

If the UDF input parameter is a `CHAR`, Vantage appends the correct number of pad characters or trims the correct number of pad characters so that the input string has the same number of characters as the column definition.

If the UDF input parameter is a `VARCHAR` and the column being compressed is a `CHAR` column, Vantage trims any pad characters.

- For VARCHAR or VARBYTE columns, Vantage invokes the UDF as if the input parameter was defined with the length specified in the column definition.

Consider the following table definition:

```
CREATE TABLE Descriptions
(d_ID INTEGER
,d_Data VARCHAR(2400) COMPRESS USING Compress_Data
DECOMPRESS USING Decompress_Data
);
```

You can write a C function that declares a VARCHAR(64000) input parameter and Vantage calls the function as if the function declared a VARCHAR(2400) input parameter.

## Rules for Writing UDFs that Decompress Character and Byte Data

- The scalar UDF must declare a single VARBYTE(*n*) input parameter that matches the length of the VARBYTE(*n*) return type of the corresponding UDF for compressing the data.
- The data type of the result parameter must be BYTE(*n*), VARBYTE(*n*), CHAR(*n*), or VARCHAR(*n*) and must be compatible with the data type of the table column.

The length of the column must be less than or equal to the length of the result parameter.

For CHAR and VARCHAR columns, you must use the CHARACTER SET phrase in the parameter declaration to specify a server character set that matches the server character set of the column.

**Recommendation:** Declare the result parameter with a data type of VARBYTE(*n*) or VARCHAR(*n*). This is best for most situations.

- If the UDF is for decompressing previously compressed character data, the UDF must call FNC\_GetOutputBufferSize to determine the maximum length of the string to build.

If the UDF return parameter is a CHAR, the UDF must append the necessary pad characters to the result to fill up the buffer. If the column type is VARCHAR, Vantage trims any pad characters.

If the UDF return parameter is a VARCHAR and the column type is CHAR, Vantage appends any necessary pad characters.

## Rules for Writing UDFs that Compress/Decompress TIME, TIMESTAMP, Period, ARRAY, and non-LOB UDT Data

Follow these guidelines to write scalar UDFs that compress/decompress column data which have the following data types: TIME, TIMESTAMP, Period, ARRAY, or supported distinct non-LOB UDT.

- The compress UDF should have the following signature:

- The UDF must declare a single input parameter which has a data type of TIME, TIMESTAMP, Period, ARRAY or supported distinct non-LOB UDT.
- The parameter data type and column data type should be the same.
- The return type should be VARBYTE(64000).
- The decompress UDF should have the following signature:
  - The UDF must declare a single VARBYTE(64000) input parameter.
  - The output of the decompress UDF should be identical to the column type.

## Rules for Writing UDFs that Compress/Decompress LOB Data

- The compress UDF should have the following signature:
  - The scalar UDF must declare a single input parameter which has a data type of BLOB, CLOB, or supported distinct LOB-type UDT.
  - The parameter data type and column data type should be the same.
  - The return type should be BLOB.
- The decompress UDF should have the following signature:
  - The scalar UDF must declare a single BLOB input parameter. The length of the input parameter should be equal to or greater than the length of the BLOB data type returned by the compression UDF.
  - The output of the decompress UDF should be identical to the column type.

## Compressing BLOB and CLOB Types

For information on UDFs with LOB input and return parameters, see [Defining Functions that Use LOB Types](#).

When writing a scalar UDF that compresses BLOB or CLOB types, use the following general structure:

1. Define a buffer size (no larger than 64KB) for reading portions of the source LOB.
2. Use FNC\_LobOpen to open reading of the source LOB.
3. While more data exists in the LOB:
  - a. Use FNC\_LobRead to read a portion of the LOB into the allocated buffer.
  - b. Perform the compression on the data in the buffer.
  - c. If the UDF cannot compress the input string, return an SQLSTATE of 'U1005', otherwise, append the compressed contents of the buffer to the output LOB using FNC\_LobAppend.
4. Use FNC\_LobClose to close reading of the source LOB.
5. Handle errors reported by any FNC routines during processing.
6. Release allocated resources once you have processed the data.

For information about the FNC functions, see [LOB Access](#).

To see sample code for compressing CLOB data, see [C Scalar Function for Compressing CLOB Data](#).

## Decompressing BLOB and CLOB Types

For information on UDFs with LOB input and return parameters, see [Defining Functions that Use LOB Types](#).

When writing a scalar UDF that decompresses BLOB or CLOB types, use the following general structure:

1. Define a buffer size (no larger than 64KB) for reading portions of the source LOB.
2. Use FNC\_LobOpen to open reading of the source LOB.
3. While more data exists in the LOB:
  - a. Use FNC\_LobRead to read a portion of the LOB into the allocated buffer.
  - b. Perform the decompression on the data in the buffer.
  - c. Append the uncompressed contents of the buffer to the output LOB using FNC\_LobAppend.
4. Use FNC\_LobClose to close reading of the source LOB.
5. Handle errors reported by any FNC routines during processing.
6. Release allocated resources once you have processed the data.

For information about the FNC functions, see [LOB Access](#).

To see sample code for decompressing CLOB data, see [C Scalar Function for Decompressing CLOB Data](#).

## Compressing Distinct UDT LOB Types

For information on UDFs with UDT input and LOB return parameters, see [Defining Functions that Use UDT Types](#) and [Defining Functions that Use LOB Types](#).

When writing a scalar UDF that compresses supported distinct UDT LOB types, follow the instructions for [Compressing BLOB and CLOB Types](#), except you must call FNC\_GetDistinctInputLob to get a LOB\_LOCATOR before you call FNC\_LobOpen to open reading of the source LOB.

For information about the FNC functions, see [UDT Interface](#) and [LOB Access](#).

To see sample code for compressing distinct UDT LOB types, see [C Scalar Function for Compressing Distinct UDT LOB Types](#).

## Decompressing Distinct UDT LOB Types

When writing a scalar UDF that decompresses supported distinct UDT LOB types, follow the instructions for [Decompressing BLOB and CLOB Types](#), except you must call FNC\_GetDistinctResultLob to get a LOB\_RESULT\_LOCATOR before you call FNC\_LobOpen to open reading of the source LOB.

## Related Information

FOR more information on ...	SEE ...
UDFs with UDT input and LOB return parameters	<ul style="list-style-type: none"> <li><a href="#">Defining Functions that Use UDT Types</a>.</li> </ul>



FOR more information on ...	SEE ...
	<ul style="list-style-type: none"> <li>• <a href="#">Defining Functions that Use LOB Types.</a></li> </ul>
UDT interface and LOB access FNC functions	<ul style="list-style-type: none"> <li>• <a href="#">UDT Interface.</a></li> <li>• <a href="#">LOB Access.</a></li> </ul>
library functions that UDFs for algorithmic compression can use for string argument and result processing	<a href="#">C Library Functions.</a>
examples that show how to implement scalar UDFs for algorithmic compression	<ul style="list-style-type: none"> <li>• <a href="#">C Scalar Function for Compressing CLOB Data.</a></li> <li>• <a href="#">C Scalar Function for Decompressing CLOB Data.</a></li> <li>• <a href="#">C Scalar Function for Compressing Distinct UDT LOB Types.</a></li> <li>• <a href="#">C Scalar Function for Decompressing Distinct UDT LOB Types.</a></li> </ul>
using COMPRESS when defining or modifying columns in a table	<p>CREATE TABLE and ALTER TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</p> <p>COMPRESS and DECOMPRESS phrases in <i>Teradata Vantage™ - Data Types and Literals</i>, B035-1143.</p>

## Defining Functions for Row Level Security

### About Row Level Security

Teradata provides row level security as one way of controlling access to tables and views. Row level security controls user access by row and by SQL operation. Access rules are based on the relationship between the user access level and the row security level.

The following steps show the basic process for implementing row level security:

1. Determine security classifications that will be used to define security labels for users and data rows.
2. Create security constraint UDFs to define and enforce row level security restrictions.
3. Create security constraint objects and specify the appropriate security UDF to control an INSERT, SELECT, UPDATE or DELETE operation. For more information, see the information about CREATE CONSTRAINT in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.
4. Assign security constraints to tables and users.

For more information about row level security, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

### Security Constraint UDFs

A security constraint UDF defines and enforces the rules that determine whether to allow execution of an INSERT, SELECT, UPDATE, or DELETE statement on a row of a table that is defined with a security constraint.

When the SQL statement is executed, Vantage runs the associated security UDF to verify if the requesting user has the access level required to perform the operation as compared to the security level assigned to the row.

If the requesting user does not have the required access level, the UDF denies the request and request processing moves on to the next applicable row. If a requesting user has the `OVERRIDE` privilege for an SQL operation, Vantage bypasses the UDF that restricts the operation. If the security constraint object does not specify a UDF for an SQL operation, the operation succeeds only if the user has the corresponding `OVERRIDE` privilege.

In normal usage, users do not directly call security constraint UDFs in an SQL statement. Instead, Vantage automatically calls the required functions when an `INSERT`, `SELECT`, `UPDATE`, or `DELETE` statement is executed against a table in which a security constraint has been defined. Therefore, you do not need to grant `EXECUTE FUNCTION` privileges on security constraint UDFs to users who must access the protected tables. However, if you are a security administrator or you are developing a security constraint UDF and want to test the return value of the constraint function, you can explicitly call the constraint function in a `SELECT` statement. In this case, you must have `EXECUTE FUNCTION` privilege on the constraint function.

## Rules for Security Constraint UDFs

The following rules apply to security constraint UDFs:

- The function must be a scalar UDF written in C or C++.
- The UDF cannot be an overloaded function.
- The UDF must reside in the `SYSLIB` database.
- If the security constraint object is defined with the `NULL` option, then the parameter style must be `SQL` to allow nulls. If the constraint object does not allow nulls, then the parameter style must be `TD_GENERAL`.
- You must drop the security constraint object before you can drop any security UDFs associated with the constraint. However, you can alter or replace the UDF as long as the replaced UDF contains the required parameters for the constraint definition.

## INSERT UDF Parameter List

`INSERT` UDFs implement the security policy for an `INSERT` operation. The parameter list for an `INSERT` UDF consists of:

- An input parameter that takes the current session security label as the argument.
- A result parameter that returns the current session security label if the `INSERT` request passed the security policy; otherwise, it returns 0. For details about the valid return values, see [Return Values for INSERT or UPDATE UDFs](#). The return data type must be the same as the type specified in the definition of the constraint object.
- Indicator parameters for the input and result parameters. These are required only if the constraint object allows nulls.

Here is an example of how to declare an INSERT UDF:

```

/***** C source file name: insertlevel.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void InsertLevel( short int  *sess_level,
                  short int  *new_row )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION SYSLIB.InsertLevel( current_session SMALLINT )
    RETURNS SMALLINT
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!insertlevel!cctests/insertlevel.c';

```

where:

- `Current_session` is the system-defined parameter name that identifies the security label currently set for the session.
- The RETURNS *data\_type* is the same as the data type specified in the definition of the constraint object.
- PARAMETER STYLE is SQL if the constraint object allows nulls; otherwise, PARAMETER STYLE is TD\_GENERAL.

## UPDATE UDF Parameter List

UPDATE UDFs implement the security policy for an UPDATE operation. The parameter list for an UPDATE UDF consists of:

- An input parameter that takes the current session security label as the argument.
- An input parameter that takes the security label from the target row as the argument.
- A result parameter that returns the current session security label if the UPDATE request passed the security policy; otherwise, it returns 0. For details about the valid return values, see [Return Values for INSERT or UPDATE UDFs](#). The return data type must be the same as the type specified in the definition of the constraint object.
- Indicator parameters for the input and result parameters. These are required only if the constraint object allows nulls.

Here is an example of how to declare an UPDATE UDF:

```

/***** C source file name: updatelevel.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void UpdateLevel( short int  *sess_level,
                  short int  *curr_row,
                  short int  *upd_row )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION SYSLIB.UpdateLevel( current_session SMALLINT,
                                   input_row          SMALLINT )
    RETURNS SMALLINT
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!updatelevel!cctests/updatelevel.c';

```

where:

- `Current_session` is the system-defined parameter name that identifies the security label currently set for the session.
- `Input_row` is the system-defined parameter name that identifies the security label from the target row.
- The `RETURNS data_type` is the same as the data type specified in the definition of the constraint object.
- `PARAMETER STYLE` is SQL if the constraint object allows nulls; otherwise, `PARAMETER STYLE` is `TD_GENERAL`.

## DELETE UDF Parameter List

DELETE UDFs implement the security policy for a DELETE operation. The parameter list for a DELETE UDF consists of:

- An input parameter that takes the security label from the target row as the argument.
- A result parameter that returns 'T' if the DELETE request passed the security policy or 'F' if the request failed the security policy. For details about the valid return values, see [Return Values for SELECT or DELETE UDFs](#).

- Indicator parameters for the input and result parameters. These are required only if the constraint object allows nulls.

Here is an example of how to declare a DELETE UDF:

```

/***** C source file name: deletelevel.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void DeleteLevel( short int  *curr_row,
                  char       *result )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION SYSLIB.DeleteLevel( input_row SMALLINT )
    RETURNS CHAR
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!deletelevel!cctests/deletelevel.c';

```

where:

- input\_row is the system-defined parameter name that identifies the security label from the target row.
- PARAMETER STYLE is SQL if the constraint object allows nulls; otherwise, PARAMETER STYLE is TD\_GENERAL.

## SELECT UDF Parameter List

SELECT UDFs implement the security policy for a SELECT operation. The parameter list for a SELECT UDF consists of:

- An input parameter that takes the current session security label as the argument.
- An input parameter that takes the security label from the target row as the argument.
- A result parameter that returns 'T' if the SELECT request passed the security policy or 'F' if the request failed the security policy. For details about the valid return values, see [Return Values for SELECT or DELETE UDFs](#).
- Indicator parameters for the input and result parameters. These are required only if the constraint object allows nulls.

Here is an example of how to declare a SELECT UDF:

```

/***** C source file name: readlevel.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void ReadLevel( short int  *sess_level,
                short int  *curr_row,
                char        *result )
{
    ...
}

```

The corresponding CREATE FUNCTION statement looks like this:

```

CREATE FUNCTION SYSLIB.ReadLevel( current_session SMALLINT,
                                input_row          SMALLINT )
    RETURNS CHAR
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!readlevel!cctests/readlevel.c';

```

where:

- `current_session` is the system-defined parameter name that identifies the security label currently set for the session.
- `input_row` is the system-defined parameter name that identifies the security label from the target row.
- `PARAMETER STYLE` is `SQL` if the constraint object allows nulls; otherwise, `PARAMETER STYLE` is `TD_GENERAL`.

## Security Constraint UDF Body

INSERT, UPDATE, DELETE and SELECT UDFs take the current session security label and/or the target row security label input arguments and compare these security values to the security policy rules defined in the UDF to determine if the INSERT, UPDATE, DELETE or SELECT should be allowed.

The security constraint UDFs return a true or false result depending on whether or not the input values pass the security policy. INSERT and UPDATE UDFs also return the current session security label if the SQL request passes the security policy. For details about the valid return values of security constraint UDFs, see [Return Values for INSERT or UPDATE UDFs](#) and [Return Values for SELECT or DELETE UDFs](#).

The following example security policy rules provide a guide for creating security constraint UDFs. The rules are based on the Bell-Lapadula Model that is commonly used for enforcement of access control in government and military applications.

No Read Up (for SELECT operations):

- The session hierarchical level must be  $\geq$  the row hierarchical level.  
Users cannot read data with a higher classification.
- The session label must include all non-hierarchical compartments found in the row label.  
The user must be assigned to all compartments under which the row is classified.

No Write Down (INSERT/UPDATE operations)

- The row hierarchical level must be  $\geq$  the session hierarchical level.  
New or updated rows inherit the session level. This rule prevents an updating user from accidentally reclassifying the row to a higher level.
- The row label must include all non-hierarchical compartments in the session label.  
New or updated rows inherit all session compartments. This rule prevents an updating user from accidentally adding excess compartmental classifications to a row.

---

**Note:**

The sample rules do not contain recommendations for a DELETE policy, but it is common to require that a row be declassified, that is, set to the lowest classification level or to NULL before it can be deleted.

---

For more information about security classification categories and labels, see *Teradata Vantage™ - Advanced SQL Engine Security Administration*, B035-1100.

## Example: Security Constraint Hierarchical Rules

Operation	Example Rule
INSERT	<p>The current session must have a security label valid for the security constraint. The session label is entered as the constraint column value for the new row.</p> <p><b>Purpose:</b> Allows any user with table level insert privileges to insert a new row. Assumes that the user security level is appropriate for the new data.</p>
SELECT	<p>The session security label must equal or exceed the row label or the operation fails.</p> <p><b>Purpose:</b> Only users classified equal to or higher than a row can read row data.</p>
UPDATE	<p>The session security label must equal or exceed the row label or the operation fails. If the operation is allowed, the updated row uses the session security label for the constraint column value.</p> <p><b>Purpose:</b> Restricts access to equivalent/higher level users. The row is automatically reclassified to the user level in case the user adds data with a higher classification.</p>
DELETE	<p>The row cannot be deleted unless the constraint column value is at the lowest security level.</p> <p><b>Purpose:</b> Ensures that a row is reviewed and declassified before it can be deleted.</p>

## Example: Security Constraint Non-Hierarchical Rules

Operation	Example Rule
INSERT	The current session must have a security label (1 or more compartments). All compartments in the session label are entered as the row constraint column value. <b>Purpose:</b> Forces predictable row classification based on the user label.
SELECT	The session security label must include all the compartments in the row label or the operation fails. <b>Purpose:</b> If a row is classified with several compartments, ensures that the accessing user is a member of all compartments.
UPDATE	The row label must include all the compartments contained in the session label. <b>Purpose:</b> Prevents the user from inadvertently adding classifications to the row.
DELETE	The row can be deleted only if the constraint column value is NULL. <b>Purpose:</b> Ensures that a row is reviewed and declassified before it can be deleted. <b>Note:</b> You must have OVERRIDE UPDATE privileges to reclassify a row as NULL, so that it can be deleted.

## Return Values for INSERT or UPDATE UDFs

INSERT and UPDATE UDFs should return the following values in the result and corresponding indicator parameters. The return values are based on the data type and NULL option defined for the constraint object. The data type for the return value must be the same as the type specified in the constraint object.

- If the constraint data type is SMALLINT and the constraint does not allow nulls, then the valid return values are:

Return Value	Description
0	Indicates that the call failed to pass the security policy. The INSERT or UPDATE is not performed and the system will move to the next row.
non-zero	Indicates that the call passed the security policy. The UDF returns the current session security label which the system will place in the constraint column of the target row. The INSERT or UPDATE operation is performed.

- If the constraint data type is SMALLINT and the constraint allows nulls, then the valid return values are:



IF the Return Indicator Value is...	AND the Return Value is...	THEN...
-1	not applicable	the call passed the security policy and the system will place a NULL in the constraint column of the target row. The INSERT or UPDATE operation is performed.
0	0	the call failed to pass the security policy. The INSERT or UPDATE is not performed and the system will move to the next row.
0	non-zero	the call passed the security policy. The UDF returns the current session security label which the system will place in the constraint column of the target row. The INSERT or UPDATE operation is performed.

- If the constraint data type is BYTE(*n*) and the constraint does not allow nulls, then the valid return values are:

Return Value where. ..	Description
all bytes are 0	Indicates that the call failed to pass the security policy. The INSERT or UPDATE is not performed and the system will move to the next row.
some byte is non-zero	Indicates that the call passed the security policy. The UDF returns the current session security label which the system will place in the constraint column of the target row. The INSERT or UPDATE operation is performed.

- If the constraint data type is BYTE(*n*) and the constraint allows nulls, then the valid return values are:

IF the Return Indicator Value is...	AND the Return Value is...	THEN...
-1	not applicable	the call passed the security policy and the system will place a NULL in the constraint column of the target row. The INSERT or UPDATE operation is performed.
0	all bytes are 0	the call failed to pass the security policy. The INSERT or UPDATE is not performed and the system will move to the next row.
0	some byte is non-zero	the call passed the security policy. The UDF returns the current session security label which the system will place in the constraint column of the target row. The INSERT or UPDATE operation is performed.

## Return Values for SELECT or DELETE UDFs

SELECT and DELETE UDFs should return the following values in the result and corresponding indicator parameters. The return values are based on the data type and NULL option defined for the constraint object.

- If the constraint data type is SMALLINT or BYTE(*n*) and the constraint does not allow nulls, then the valid return values are:

Return Value	Description
'T'	Indicates that the call passed the security policy. The SELECT or DELETE operation is performed.
'F'	Indicates that the call failed to pass the security policy. The SELECT or DELETE is not performed and the system will move to the next row.

- If the constraint data type is SMALLINT or BYTE(*n*) and the constraint allows nulls, then the valid return values are:

IF the Return Indicator Value is...	AND the Return Value is...	THEN...
0	'T'	the call passed the security policy. The SELECT or DELETE operation is performed.
0	'F'	the call failed to pass the security policy. The SELECT or DELETE operation is not performed and the system will move to the next row.

### Note:

The UDF cannot return -1 as the value for the return indicator parameter. This is an invalid result.

## General Global Functions

If you are creating a set of UDFs that perform similar operations, chances are good that the UDFs are using some common functions. Instead of repeating the common functions as internal (static) functions for each UDF, you can create a set of general global functions that can be used by UDFs in the same database.

## Functions Available at Database Level

General global functions are available to any UDF in the database in which they are created, because all the UDFs that are defined in a specific database are linked into a single dynamically linked library.

## Creating General Global Functions

One way to create a general global function is to supply the code in a C/C++ function for the first UDF you create.

For example, suppose you have the following general global function:

```

/***** Source code filename: global1.c *****/
int global1( char *text1)
{
    ...
}

```

The C code for the first UDF declares the general global function as external, and looks like this:

```

/***** Source code filename: udf1.c *****/
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

external global1(char *text1);

void udf1(Character *a, INTEGER *result,
        ... )
{
    int global_result;

    global_result = global1(a);
    ...
}

```

The corresponding CREATE FUNCTION statement must specify the source for the general global function and the first UDF:

```

CREATE FUNCTION UDF1(A CHAR(30) CHARACTER SET LATIN)
RETURNS INTEGER
...
EXTERNAL NAME 'CS!global1!global1.c!CS!udf1!udf1.c';

```

## Using the General Global Function in Other UDFs

Other UDFs in the same database that use the global1 code declare the general global function as external:

```

/***** Source code filename: udf2.c *****/
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

external global1(char *text1);

void udf2(VARCHAR *b, INTEGER *result,
        ... )
{
    int global_result;

    global_result = global1(b);

    ...
}

```

The corresponding CREATE FUNCTION statement only has to specify the source for the UDF:

```

CREATE FUNCTION UDF1(B VARCHAR(20000) CHARACTER SET LATIN)
RETURNS INTEGER
...
EXTERNAL NAME 'CS!udf2!udf2.c';

```

The code for the general global function only had to be provided once and can be shared by all other UDFs in the same library as long as they provide the external statement.

## Installing a Library of General Global Functions

Another way to provide general global functions that UDFs can reference is to install a separate library of functions on the database on all nodes.

Here is an example that uses the package name clause of the CREATE FUNCTION statement to install a library of functions:

```

CREATE FUNCTION UDF1(A CHAR(30))
RETURNS INTEGER
...
EXTERNAL NAME 'SP!/usr/local/lib/libGenUdfs.so!F!udf1'

```

where `libGenUdfs.so` is a shared object file containing the library of functions that *UDF1* can call and *udf1* is the function entry point name in the shared object file for *UDF1*.

The shared object file must already be installed on the system and distributed to all its nodes before you use the CREATE FUNCTION statement. If you migrate to another system, you must copy the shared object file to /usr/local/lib on the target system before starting the database migration.

For more information on installing libraries, see [Administration](#).

## Dropping General Global Functions

You cannot drop the first UDF that also contains the general global function if there is a dependency by other UDFs in the library.

If there is such a dependency, you can drop UDFs in reverse order from which they were created.

If the entire database is deleted, then the entire UDF DLL or SO is deleted without regard to any dependencies existing between UDFs.

## C/C++ External Stored Procedures

In addition to stored procedures, which use SQL control and condition-handling statements, input and output parameters, and local variables to provide applications with a server-based, precompiled procedural interface, Vantage supports external stored procedures.

You can write your own external stored procedures in the C or C++ programming language, install them on the database, and then use the SQL CALL statement to call them like other stored procedures. An external stored procedure runs on a PE.

C and C++ external stored procedures can use Call-Level Interface Version 2 (CLLv2, also referred to as CLI) to directly execute SQL statements. C and C++ external procedures that do not use CLLv2 can make a C library call to invoke a stored procedure and execute SQL statements indirectly.

This section uses the term *stored procedure* to refer to a stored procedure that is written using SQL statements and the term *external stored procedure* to refer to a stored procedure that is written using C or C++.

For more information on writing external stored procedures in the Java programming language, see [Java External Stored Procedures](#).

### Overall Development Synopsis

The steps you take to develop, compile, install, and invoke an external stored procedure depend on whether the external stored procedure issues operating system I/O calls and whether the external stored procedure executes SQL. The meaning of I/O as it applies to external stored procedures is any operating system call that requires the operating system to retain a resource context, such as for open files or other operating system services. Such resource usage usually returns a handle to the caller that is used to access the resource and must be released when finished using it.

The next few pages provide high level overviews of the steps you take to develop external stored procedures. You can find details for each step in the following sections.

### External Stored Procedures That Do Not Perform I/O or Execute SQL

Here is a synopsis of the steps you take to develop, compile, install, and invoke a C or C++ external stored procedure that does not use CLLv2 and does not perform I/O:

1. Use CREATE PROCEDURE or REPLACE PROCEDURE to identify the location of the source code, object, or package, and install it on a development or test database.

**Recommendation:** In general, you should not create external stored procedures in Teradata system databases such as SYSLIB or SYSUDTLIB. For details, see [Installing a C/C++ External Stored Procedure](#).

The external stored procedure is compiled, if the source code is submitted, linked to the dynamic linked library (DLL or SO) associated with the database in which the external stored procedure resides, and distributed to all database nodes in the system.

2. Test and debug the external stored procedure in Vantage in *protected* execution mode until you are satisfied it works correctly.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions for the database. For more information, see [C/C++ Command-line Debugging for UDFs](#).

Protected mode is the default execution mode for an external stored procedure. In protected mode, Vantage isolates all of the data the procedure might access as a separate process in its own local workspace. If any memory violation or other system error occurs, the error is localized to the procedure and the transaction executing the procedure. This makes the procedure run slower.

3. Use ALTER PROCEDURE to change the external stored procedure to run in nonprotected execution mode.
4. Rerun the tests from Step 3 to test the external stored procedure in nonprotected execution mode until you are satisfied it works correctly.
5. Install the external stored procedure on your production system.
6. Use GRANT to grant privileges to users who are authorized to use the external stored procedure.

## External Stored Procedures that Execute SQL

Here is a synopsis of the steps you take to develop, compile, install, and invoke a C or C++ external stored procedure that uses CLIV2 to execute SQL:

1. Write, test, and debug the C or C++ code for the external stored procedure.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions for the database. For more information, see [C/C++ Command-line Debugging for UDFs](#).

2. If the external stored procedure requires access to specific operating system resources, use CREATE AUTHORIZATION or REPLACE AUTHORIZATION to create a context that identifies a native operating system user and allows external stored procedures to perform I/O by running as separate processes under the authorization of that user.
3. Use CREATE PROCEDURE or REPLACE PROCEDURE with options that provide specific information about the external stored procedure.

Option	Description
<ul style="list-style-type: none"> <li>• NO SQL (Default)</li> <li>• CONTAINS SQL</li> <li>• READS SQL DATA</li> <li>• MODIFIES SQL DATA</li> </ul>	Indicates whether the external stored procedure can execute SQL statements and whether those statements can read or modify SQL data in the database.

Option	Description
<ul style="list-style-type: none"> <li>LANGUAGE C</li> <li>LANGUAGE CPP</li> </ul>	Identifies the source code language of the external stored procedure.
<ul style="list-style-type: none"> <li>PARAMETER STYLE SQL (Default)</li> <li>PARAMETER STYLE TD_GENERAL</li> </ul>	Indicates whether the external stored procedure can accept null IN or INOUT arguments or return null IN or INOUT arguments.
DYNAMIC RESULT SETS (Optional)	Specifies the number of result sets that the external stored procedure returns.
EXTERNAL NAME	Provides the location of the source code and specifies the Teradata CLI package name.
EXTERNAL SECURITY (Optional)	Associates execution of the external stored procedure with the context created by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement in the previous step.

**Recommendation:** In general, you should not create external stored procedures in Teradata system databases such as SYSLIB or SYSUDTLIB. For details, see [Installing a C/C++ External Stored Procedure](#).

The external stored procedure is compiled, linked to a CLI-specific external stored procedure dynamic linked library (DLL or SO) associated with the database in which the external stored procedure resides, and distributed to all database nodes in the system.

4. Test the external stored procedure until you are satisfied it works correctly.
5. Use GRANT to grant privileges to users who are authorized to use the external stored procedure.

## External Stored Procedures that Issue Operating System I/O Calls and Do Not Execute SQL

Here is a synopsis of the steps you take to develop, compile, install, and invoke an external stored procedure that issues operating system I/O calls:

1. Write, test, and debug the C or C++ code for the external stored procedure.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions for the database. For more information, see [C/C++ Command-line Debugging for UDFs](#).

2. Determine the level of access to operating system services that the external stored procedure requires.

IF the external stored procedure ...	THEN ...
accesses operating system resources that ordinary	the external stored procedure can run in protected execution mode as a separate process under 'tdatuser', a local operating system user that the Vantage installation process creates.



IF the external stored procedure ...	THEN ...
operating system users have access to	
requires access to specific operating system resources	use CREATE AUTHORIZATION or REPLACE AUTHORIZATION to create a context that identifies a native operating system user and allows external stored procedures to perform I/O by running as separate processes under the authorization of that user.

- Use CREATE PROCEDURE or REPLACE PROCEDURE to identify the location of the source code, object, or package, and install it on a development or test database.

**Recommendation:** In general, you should not create external stored procedures in Teradata system databases such as SYSLIB or SYSUDTLIB. For details, see [Installing a C/C++ External Stored Procedure](#).

IF you ...	THEN the CREATE PROCEDURE or REPLACE PROCEDURE statement ...
did not use CREATE AUTHORIZATION or REPLACE AUTHORIZATION in the previous step	sets the default execution mode for the external stored procedure to protected mode. The external stored procedure runs under the tdatuser operating system user and can access the system resources for which tdatuser has privileges.
used CREATE AUTHORIZATION or REPLACE AUTHORIZATION in the previous step	must specify the EXTERNAL SECURITY clause to associate execution of the external stored procedure with the context created by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement. The external stored procedure runs under the operating system user identified by the specified context and can access the system resources for which the user has privileges.

The external stored procedure is compiled, if the source code is submitted, linked to the dynamic linked library (DLL or SO) associated with the database in which the external stored procedure resides, and distributed to all database nodes in the system.

- Test the external stored procedure on the development or test system until you are satisfied it works correctly.
- Install the external stored procedure on your production system.
- Use GRANT to grant privileges to users who are authorized to use the external stored procedure.

## Related Information

FOR more information on ...	SEE ...
creating an external stored procedure	related topics in this document.

FOR more information on ...	SEE ...
debugging an external stored procedure	<ul style="list-style-type: none"> <li>• <a href="#">Debugging an External Stored Procedure.</a></li> <li>• <a href="#">C/C++ Command-line Debugging for UDFs.</a></li> </ul>
<ul style="list-style-type: none"> <li>• CREATE PROCEDURE</li> <li>• REPLACE PROCEDURE</li> </ul>	<ul style="list-style-type: none"> <li>• <a href="#">Installing a C/C++ External Stored Procedure.</a></li> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
<ul style="list-style-type: none"> <li>• CREATE AUTHORIZATION</li> <li>• REPLACE AUTHORIZATION</li> </ul>	<ul style="list-style-type: none"> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
code examples for external stored procedures	<a href="#">External Stored Procedure Code Examples.</a>

## Source Code Development for C/C++ External Stored Procedures

You can develop the body of an external stored procedure using the C or C++ programming language.

### Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

## Source Code Contents

The topics that follow provide details about the contents of the C or C++ source code for an external stored procedure. In general, the contents of the C or C++ source code must:

- Define the SQL\_TEXT constant
- Include the sqltypes\_td.h header file
- Define the parameter list in the order that matches the parameter passing convention specified in the CREATE PROCEDURE statement

### Note:

A C++ UDF or external procedure must catch all C++ exceptions by the end of the UDF or external procedure code. Any exception that is not caught and is thrown out from the routine may result in unexpected behavior.

## Internal and External Data

All data is local to the procedure. No global data can be retained from procedure call to procedure call, with the exception of GLOP data. For details on GLOP data, see [Global and Persistent Data](#).

A procedure cannot contain any static variables. A procedure can contain static constants.

You can specify the External Data Access clause in the CREATE PROCEDURE/REPLACE PROCEDURE statement to define the relationship between a function or external stored procedure and data that is external to Vantage.

The option you specify determines:

- Whether or not the external routine can read or modify external data
- Whether or not the database will redrive the request involving the function or procedure after a database restart

If Redrive protection is enabled, the system preserves responses for completed SQL requests and resubmits uncompleted requests when there is a database restart. However, if the External Data Access clause of an external routine is defined with the MODIFIES EXTERNAL DATA option, then the database will not redrive the request involving that function or procedure. For details about Redrive functionality, see:

- *Teradata Vantage™ - Database Administration*, B035-1093
- The RedriveProtection and RedriveDefaultParticipation DBS Control fields in *Teradata Vantage™ - Database Utilities*, B035-1102.

If you do not specify an External Data Access clause, the default is NO EXTERNAL DATA.

The following table explains the options for the External Data Access clause and how the database uses them for external routines.

Option	Description
MODIFIES EXTERNAL DATA	The routine modifies data that is external to the database. In this case, the word <i>modify</i> includes delete, insert, and update operations.  <b>Note:</b> Following a database failure, the database does not redrive requests involving a function or external stored procedure defined with this option.
NO EXTERNAL DATA	The routine does not access data that is external to the database. This is the default.
READS EXTERNAL DATA	The routine reads, but does not modify data that is external to the database.

For more information, see the information about CREATE PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## I/O

External stored procedures cannot do standard I/O or network I/O.

Under the following conditions, external stored procedures can do I/O or otherwise interface with the operating system such that the operating system retains resources such as file handles or object handles.

IF the CREATE PROCEDURE or REPLACE PROCEDURE statement ...	THEN the external stored procedure ...
specifies the EXTERNAL SECURITY clause	<p>executes as a separate process under the authorization of a specific native operating system user established by a CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement.</p> <p>The external stored procedure can access the system resources for which the user has privileges.</p> <p>The procedure must release opened resources (close handles) before it completes.</p>
omits the EXTERNAL SECURITY clause	<p>can do I/O if the external stored procedure runs in protected execution mode. In protected execution mode, the external stored procedure runs as a separate process under 'tdatuser', a local operating system user that the Vantage installation process creates.</p> <p>The external stored procedure can access the system resources for which tdatuser has privileges.</p> <p>The procedure must release opened resources (close handles) before it completes.</p> <p>For more information on protected execution mode, see <a href="#">Protected Mode Execution</a>.</p>

## Signal Handling

External stored procedures that run on Linux systems cannot use or modify the handling of the signal SIGUSR2, which is reserved by Vantage.

## Standard C Library Functions

An external stored procedure can call standard C library functions that do not do any I/O, such as string library functions.

Restrictions apply to some functions, such as *malloc* and *free*.

For more information, see [Using Standard C Library Functions](#).

## Teradata Library Functions

Teradata provides functions that you can use for external stored procedure development. For example, you can use the FNC\_DbsInfo function to get session information related to the current execution of a procedure.

The following table shows some of the categories of library functions that Teradata provides.

Category	Use
ARRAY Data Type Interface	Enable an external stored procedure to access and set the values of the elements in an ARRAY data type
Global Information	Returns session information related to the current execution of an external stored procedure
Lob Access	Enable an external stored procedure to use a locator to access the contents of a referenced object and to append data to a LOB object
Period Data Type Interface	Enable an external stored procedure to access and set the value of a Period data type
Query Band Access	Return query band name-value pairs that have been set on a session, transaction, or profile to identify the originating source of queries and help manage task priorities and track system use
Stored Procedure Invocation	Provides a way to call a stored procedure from an external stored procedure
String Argument and Result Processing	Useful for processing BYTE, CHAR, or VARCHAR parameters
TD_ANYTYPE Parameter Access	Enable an external stored procedure to retrieve information about TD_ANYTYPE IN, INOUT, and OUT parameters
Trace	Let you get trace output for debugging purposes during external stored procedure development
UDT Interface	Enable an external stored procedure to access and set the value of a distinct UDT or attribute values of a structured UDT

For a list of all available functions and their descriptions, see [C Library Functions](#).

## Problems Using System V IPC and POSIX IPC

Teradata recommends that UDFs, UDMs, and external stored procedures not use System V IPC or POSIX IPC such as semaphore, mutex, conditional variable, and shared memory. The Teradata C library does not provide FNC functions to allocate and deallocate these IPC resources. When a session aborts or unexpected events cause Teradata to terminate the external routine execution threads or processes, there is a risk that these IPC resources may be left over in the system without being cleaned up. In rare cases, this may cause a system hang or OS resource leak.

## Consequences of Using the exit() System Call

Avoid calling `exit()` from external stored procedures.

External stored procedures that call `exit()` generate the following results:

- The request that called the external stored procedure is rolled back.
- The server process is terminated and must be recreated for the next external stored procedure, causing additional overhead.

## C/C++ Header Files

Teradata provides the `sqltypes_td.h` header file that you include in your source code file. The header file defines the equivalent C data types for all database data types that you can use for the input arguments and result of your external stored procedures. Every SQL data type corresponds to a C data type in `sqltypes_td.h`.

## Location

The header file is in the `etc` directory of the Teradata software distribution:

```
/usr/tdbms/etc
```

To verify the path of the `etc` directory, enter the following on the command line:

```
pdepath -e
```

## SQL\_TEXT Definition

Before you include the `sqltypes_td.h` header file, you must define the `SQL_TEXT` constant. The value that you use must match the current server character set of the session in which you use the `CREATE PROCEDURE` statement to create the external stored procedure.

IF you use the <code>CREATE PROCEDURE</code> statement when the current server character set is ...	THEN the C or C++ function must set <code>SQL_TEXT</code> to ...
KANJI	Kanji1_Text
KANJISJIS	Kanjisjis_Text
LATIN	Latin_Text
UNICODE	Unicode_Text

`SQL_TEXT` is used for specific arguments when the external stored procedure uses parameter style SQL. For details on parameter styles, see [External Stored Procedure Parameter List](#).

Vantage remembers the character set the procedure was created under so that the procedure can translate SQL\_TEXT input arguments to the expected text and translate text to SQL\_TEXT output arguments, no matter who calls the procedure and what the current server character set is for the session.

## NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

## Example: Defining SQL\_TEXT and Including sqltypes\_td.h

The following example shows how to define SQL\_TEXT and include the sqltypes\_td.h header file in the file that defines an external stored procedure:

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
```

Alternatively, you can use the following line to include the sqltypes\_td.h header file:

```
#include <sqltypes_td.h>
```

Using angle brackets (< >) or double quotation marks (" ") affects the path that the C preprocessor uses to search for the sqltypes\_td.h header file.

## SQL Data Types

Every SQL data type corresponds to a C data type that you use for the input arguments and result of your external stored procedures.

For details, see [SQL Data Type Mapping](#).

For exact definitions, see the sqltypes\_td.h header file.

## Including CLlv2 Header Files for External Stored Procedures with SQL

If your external stored procedure uses CLlv2 to directly execute SQL, include the CLlv2 header files after the sqltypes\_td.h header file. For example:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
```

```
#include <string.h>
#include <stdio.h>
#include <wchar.h>
#include <coptypes.h>
#include <coperr.h>
#include <parcel.h>
#include <dbcarea.h>
```

## C/C++ Function Name

The name of the function in the C or C++ source code follows the C function naming conventions, with the additional restrictions that it follows database object naming rules. For more information about object naming, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

In C++, the function definition must be preceded by extern "C" so that the function name is not converted to a C++ overloaded name. For example:

```
extern "C"
void getregion( VARCHAR_LATIN *region, char sqlstate[6])
{
    ...
}
```

The function can call any module written in C++.

When you use the CREATE PROCEDURE or REPLACE PROCEDURE statement to install the external stored procedure, you specify the name of the C or C++ function. For more information, see [Specifying the C/C++ Function Name](#).

On Linux, long names can sometimes cause errors when you install an external stored procedure. For more information, see [Argument list too long](#).

## External Stored Procedure Parameters

### Parameter List

The list of parameters for an external stored procedure is very specific. It includes the IN, OUT, and INOUT parameters that are specified when the stored procedure appears in a CALL statement. It also includes the SQLSTATE result code.

A C or C++ external stored procedure can have 0 to 256 input/output parameters. The types that you use as input/output parameters correspond to SQL data types in the procedure invocation. For more information, see [SQL Data Type Mapping](#).

Additional parameters might be required, depending on the parameter passing convention you choose.



Parameters are always specified as pointers to the data.

## Compatible Types

The argument types of an external stored procedure in a CALL statement must either be compatible with the corresponding parameter declarations in the function definition or must be types that are implicitly converted to the corresponding parameter types, according to the implicit type conversion rules.

To pass an argument that is not compatible with the corresponding parameter type and is not implicitly converted by Vantage, the CALL statement must explicitly convert the argument to the proper type.

For information on data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Parameter Passing Convention

External stored procedures support two types of parameter passing conventions.

Parameter Passing Convention	Description
Parameter Style SQL	Provides a way to pass nulls as IN or INOUT arguments and return nulls as INOUT or OUT arguments.
Parameter Style TD_GENERAL	Does not accept null IN or INOUT arguments and does not return null INOUT or OUT arguments.

The parameter passing convention you use to code an external stored procedure must correspond to the parameter passing specification in the CREATE PROCEDURE statement for the external stored procedure.

IF CREATE PROCEDURE specifies ...	THEN use the following syntax for the function parameter list ...
PARAMETER STYLE SQL or omits the PARAMETER STYLE option	<a href="#">Syntax for Parameter Style SQL.</a>
PARAMETER STYLE TD_GENERAL	<a href="#">Syntax for Parameter Style TD_GENERAL.</a>

## External Stored Procedure Parameter List

The parameter list must correspond to the parameter style in the CREATE PROCEDURE statement for the procedure.

Use parameter style SQL to allow a procedure to pass nulls for arguments; otherwise, use parameter style TD\_GENERAL.

When you use parameter style SQL, you must define indicator parameters for each parameter.

## External Stored Procedure Parameter Style TD\_GENERAL Syntax

```
void procedure_name (
    [ input_parameter_specification [...] ]
    result_type *result,
    char sqlstate[6]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

## External Stored Procedure Parameter Style SQL Syntax

```
void procedure_name (
    [ input_parameter_specification [...] ]
    result_type *result,
    [ indicator_parameter_specification [...] ]
    int indicator_result,
    char sqlstate[6]
    SQLTEXT procedure_name [m]
    SQLTEXT specific_procedure_name [l]
    SQLTEXT error_message [p]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

### *indicator\_parameter\_specification*

```
int *indicator_parameter,
```

## External Stored Procedure Parameter List Syntax Elements

### ***procedure\_name***

Pointer to a C string whose value is the procedure name in the CREATE PROCEDURE definition.

The procedure can use this name to build error messages.

### ***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE PROCEDURE definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is the C type in `sqltypes_td.h` that corresponds to the SQL data type of *input\_parameter*.

### ***result***

Pointer to a data area big enough to hold the result that the procedure returns, as defined by the RETURNS clause in the corresponding CREATE PROCEDURE statement.

### ***indicator\_parameter\_specification***

[Optional] Indicator parameter corresponding to an input parameter.

Each *input\_parameter\_specification* must have a corresponding *indicator\_parameter\_specification*. The input parameters and indicator parameters must be in the same order.

If the value of *indicator\_parameter* is -1, the value of the corresponding *input\_parameter* is null.

If the value of *indicator\_parameter* is 0, the value of the corresponding *input\_parameter* is a non-null value.

### ***indicator\_result***

Result indicator parameter corresponding to the result.

### ***sqlstate***

Pointer to a six-character C string that indicates the SQLSTATE value—success, exception, or warning. The first five characters are ASCII and the sixth is the C null character. The string is initialized to '00000', which indicates success.

For more information on SQLSTATE values, see [Returning SQLSTATE Values](#).

***m***

Number of characters in the procedure name in the CREATE PROCEDURE definition. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a procedure name.

***specific\_procedure\_name***

Pointer to a C string whose value is the name of the external procedure being invoked.

If the CREATE PROCEDURE statement includes the SPECIFIC clause, *specific\_procedure\_name* is the name in the SPECIFIC clause; otherwise, *specific\_procedure\_name* is the same as *procedure\_name*.

The procedure can use this name to build error messages.

***l***

Number of characters in the name of the external procedure. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a procedure name.

***error\_message***

Pointer to a C string whose value is the the error message text.

***p***

Number of characters in the error message text. The maximum value is 256.

**Related Information:**

[Argument Behavior](#)

**Example: Procedure with Parameter Style TD\_GENERAL**

Here is a code excerpt that shows how to declare a C function for an external stored procedure that uses parameter style TD\_GENERAL:

```

/***** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,
                   char          sqlstate[6])
{

```

```

    ...
}

```

For a complete example of the C function, see [Example: Basic External Stored Procedure](#).

The corresponding CREATE PROCEDURE statement to install the external stored procedure on the server looks like this:

```

CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE TD_GENERAL;

```

## Example: Procedure with Parameter Style SQL

Here is an example of how to declare a C function for an external stored procedure that uses parameter style SQL:

```

/***** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,
                   int           *region_isnull,
                   char           sqlstate[6],
                   SQL_TEXT       extname[129],
                   SQL_TEXT       specific_name[129],
                   SQL_TEXT       error_message[257] )
{
    ...
}

```

For a complete example of the C function, see [Example: Basic External Stored Procedure](#).

The corresponding CREATE PROCEDURE statement looks like this:

```

CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL

```

```
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE SQL;
```

## External Stored Procedure C/C++ Function Body

### Basic C/C++ Function Definition

Here are the basic steps you take to define an external stored procedure C/C++ function:

1. Define the SQL\_TEXT constant.

For more information, see [SQL\\_TEXT Definition](#).

2. Include the sqltypes\_td.h header file.

For more information, see [C/C++ Header Files](#).

3. Include other header files that define macros and variables that the function uses.

If the external stored procedure uses CLv2 to directly execute SQL, include CLv2 header files.

4. Define the function parameter list in the order that matches the parameter passing convention specified in the CREATE PROCEDURE statement.

For more information, see [External Stored Procedure Parameter List](#).

5. Implement the function and set any INOUT or OUT arguments to the appropriate value.

6. If the function detects an error, set the:

- *sqlstate* argument to an SQLSTATE exception or warning condition before the function exits.

For more information, see [Returning SQLSTATE Values](#).

- *error\_message* string to the error message text. The characters must be inside the LATIN character range. The string is initialized to a null-terminated string on input.

7. If the function uses parameter style SQL, set the indicator\_parameter arguments for the corresponding INOUT or OUT arguments.

IF the INOUT or OUT argument is ...	THEN set the corresponding indicator_parameter argument to ...
NULL	-1.
not NULL	0.

### Example: Basic External Stored Procedure

Here is an example of a simple C function that takes an INOUT string argument, strips off the first four characters, and returns the result. The external stored procedure uses parameter style TD\_GENERAL.

```

/***** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xsp_getregion( VARCHAR_LATIN *region,
                  char          sqlstate[6])
{
    char tmp_string[64];
    if (strlen((const char *)region) > 4)
    {
        /* Strip off the first four characters */
        strcpy(tmp_string, (char *)region);
        strcpy((char *)region, &tmp_string[4]);
    }
}

```

## External Stored Procedures That Use CLOB or BLOB Arguments

External stored procedures can define CLOB or BLOB input and output arguments, passing them by locator.

An external stored procedure can be called in a request that inputs inline CLOBs or BLOBs; however, an external stored procedure cannot be used in a request that returns an inline CLOB or BLOB. A workaround for this restriction is to create a stored procedure that calls the external stored procedure, which returns the CLOB or BLOB argument by locator to the stored procedure, which in turn returns the CLOB or BLOB.

The guidelines for defining external stored procedures that use CLOB or BLOB arguments are similar to the guidelines for defining UDFs that use CLOB or BLOB arguments. For details, see [Defining Functions that Use LOB Types](#).

Note that in the C source code for the external stored procedure, you must use the FNC LOB access functions to access the value of the LOB passed into or out of the procedure. This includes the following functions:

- FNC\_LobOpen
- FNC\_LobRead
- FNC\_LobAppend
- FNC\_LobClose

For details about these and other LOB-related FNC functions, see [C Library Functions](#).

The requirement for using the FNC LOB access functions also apply in order for the external stored procedure to return default LOB values.

## External Stored Procedures That Use UDT Arguments

External stored procedures can define distinct and structured UDT input and output arguments, passing them by UDT handle.

The guidelines for defining external stored procedures that use UDT arguments are similar to the guidelines for defining scalar UDFs that use UDT arguments. For details, see [Defining Functions that Use UDT Types](#).

## External Stored Procedures That Use Period Arguments

External stored procedures can define Period input and output arguments, passing them by PDT handle.

The guidelines for defining external stored procedures that use Period arguments are similar to the guidelines for defining scalar UDFs that use Period arguments. For details, see [Defining Functions that Use Period Types](#).

## External Stored Procedures That Use ARRAY Arguments

External stored procedures can define ARRAY input and output arguments, passing them by ARRAY handle.

The guidelines for defining external stored procedures that use ARRAY arguments are similar to the guidelines for defining scalar UDFs that use ARRAY arguments. For details, see [Defining Functions that Use ARRAY Types](#).

## External Stored Procedures That Use TD\_ANYTYPE Arguments

You can define external stored procedures with IN, INOUT, or OUT parameters that are of TD\_ANYTYPE data type.

The guidelines for defining external stored procedures that use TD\_ANYTYPE arguments are similar to the guidelines for defining scalar UDFs that use TD\_ANYTYPE arguments. For details, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).

When invoking an external stored procedure that is defined with a TD\_ANYTYPE OUT parameter, you can specify the RETURNS *data type* or RETURNS STYLE *column expression* clauses along with the OUT argument in the CALL statement to indicate the desired return type of the OUT parameter. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

The RETURNS or RETURNS STYLE clause is not mandatory as long as the procedure also includes a TD\_ANYTYPE input parameter. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD\_ANYTYPE IN or INOUT argument is used to determine the return type of the OUT parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.



The RETURNS and RETURNS STYLE clauses are only used to set the return type for a TD\_ANYTYPE OUT parameter. The data type of a TD\_ANYTYPE INOUT parameter is determined by the data type of the corresponding input argument.

## External Stored Procedures That Use CLv2 to Directly Execute SQL

External stored procedures can use CLv2 to directly execute SQL. For guidelines, see [Executing SQL in C/C++ External Stored Procedures](#).

## Returning SQLSTATE Values

The parameter list of an external stored procedure includes an output character string for returning the SQLSTATE result code value.

### C Data Type

The following table defines the C data type that corresponds to the SQLSTATE result code variable.

Result Code Variable	C Data Type
SQLSTATE	char sqlstate[6]

## SQLSTATE Values

The first five characters of the *sqlstate* output character string have a format of 'ccsss', where *cc* is the class and *sss* is the subclass. The last character of the string is a binary 0, or C string terminator.

For details on the valid settings that an external stored procedure can return for the SQLSTATE result code, see [SQLSTATE Values](#).

### Initial Value

The *sqlstate* output character string is initialized to '00000' (five zero characters), which corresponds to a success condition. Therefore, you do not have to set the value of the *sqlstate* output argument for a normal return.

### Display Format

If an external stored procedure returns an SQLSTATE value other than success, a BTEQ session displays an error.

IF the SQLSTATE category is ...	THEN the display format is ...
warning	<pre>*** Warning: 7505 in UDF/XSP dbname.xspname: SQLSTATE 01H xx: &lt;text&gt;</pre> <p>where:</p> <ul style="list-style-type: none"> <li>7505 is the Teradata designated warning code for user-defined functions, user-defined methods, and external stored procedures</li> <li><i>dbname.xspname</i> is the name of the external stored procedure and the name of the database in which the stored procedure resides</li> <li>01H xx is the value that the external stored procedure sets the SQLSTATE output argument to, according to values for the warning category in the table in <a href="#">SQLSTATE Values</a>.</li> <li><i>&lt;text&gt;</i> is the value of the error message output argument, if the external stored procedure uses parameter style SQL</li> </ul>
not a warning	<pre>*** Failure 7504 in UDF/XSP dbname.xspname: SQLSTATE ccsss: &lt;text&gt;</pre> <p>where:</p> <ul style="list-style-type: none"> <li>7504 is the Teradata designated error code for user-defined functions, user-defined methods, and external stored procedures</li> <li><i>dbname.xspname</i> is the name of the external stored procedure and the name of the database in which the stored procedure resides</li> <li><i>ccsss</i> is the value that the external stored procedure sets the SQLSTATE output argument to, according to the table in <a href="#">SQLSTATE Values</a>.</li> <li><i>&lt;text&gt;</i> is the value of the error message output argument, if the external stored procedure uses parameter style SQL</li> </ul>

## Value Returned to Stored Procedures

When a stored procedure calls an external stored procedure, the SQLSTATE result code value is the same value that the external stored procedure sets, not a value of 'T7505' or 'T7504' as might be expected.

## Example: Returning the SQLSTATE Result Code Value

Consider the following external stored procedure C function:

```
void xsp_getregion( VARCHAR_LATIN *region,
                  char            sqlstate[6])
{
    ...
}
```

You can use the *sqlstate* argument to return the SQLSTATE result code value.

For example, if the value of the *region* argument is not a valid value, you can set the value of the *sqlstate* argument to return a data exception:

```
strcpy(sqlstate, "U0005");
```

In a BTEQ session, the exception condition appears in the following format, where *dbname* is the name of the database of the external stored procedure:

```
*** Failure 7504 in UDF/XSP  dbname.xsp_getregion: SQLSTATE U0005:
```

## Related Information

FOR more information on ...	SEE ...
using a warning condition to debug a function	<a href="#">Forcing an SQL Warning Condition.</a>

## Calling Stored Procedures From External Stored Procedures

A C or C++ external stored procedure that does not use CLIV2 can invoke a stored procedure by calling the `FNC_CallSP` library function. Calling a stored procedure provides a way for external stored procedures to indirectly execute SQL statements.

For information on how to use CLIV2 to call a stored procedure from an external stored procedure, see [Executing SQL in C/C++ External Stored Procedures](#).

## Overall Procedure

Here is a synopsis of the steps you take in an external stored procedure to call a stored procedure:

1. Define the IN, INOUT, or OUT arguments of the same data type as the stored procedure parameters.

For example:

```
VARCHAR_LATIN  regionName[64];
INTEGER        regionCount;

strcpy((char *)regionName, "Southwest");
```

2. Define an array of pointers to the arguments to be passed to or returned from the stored procedure being called. The number of elements in the array must match the number of parameters expected by the stored procedure being called.

For example:

```
void *argv[2];

argv[0] = regionName;
```

3. Define an integer array for indicator values that indicate whether arguments passed to or returned from the stored procedure are null. The number of elements in the array must match the number of parameters expected by the stored procedure being called.

The external stored procedure sets the values corresponding to IN and INOUT arguments. The stored procedure will return values corresponding to INOUT and OUT arguments.

For example:

```
int ind[2];

ind[0] = 0; /* 0 indicates the IN or INOUT value is not null */
```

4. Define a *parm\_t* array for the data type, direction (IN, OUT, or INOUT), and attributes of each stored procedure argument. The number of elements in the array must match the number of parameters expected by the stored procedure being called.

The *parm\_t* structure is defined in `sqltypes_td.h` as:

```
typedef struct parm_t
{
    dtype_et    datatype;
    dmode_et    direction;
    charset_et  charset;
    union {
        int  length;
        int  intervalrange;
        int  precision;
        struct {
            int  totaldigit;
            int  fracdigit;
        } range;
    } size;
} parm_t;
```

IF an element in the *parm\_t* array corresponds to an argument of this type ...

THEN the following *parm\_t* members must have values ...

CHAR or VARCHAR

datatype  
direction

IF an element in the <i>parm_t</i> array corresponds to an argument of this type ...	THEN the following <i>parm_t</i> members must have values ...
	charset size.length
BYTE or CHARACTER CHARACTER SET GRAPHIC	datatype direction size.length
INTERVAL SECOND(m, n)	datatype direction size.range.totaldigit /* m */ size.range.fracdigit /* n */
any other Interval	datatype direction size.intervalrange
TIME(p) or TIMESTAMP(p)	datatype direction size.precision /* p */
DECIMAL(m, n)	datatype direction size.range.totaldigit /* m */ size.range.fracdigit /* n */
anything else	datatype direction

For example:

```
parm_t dtype[2];

dtype[0].datatype = VARCHAR_DT;
dtype[0].direction = IN_PM;
dtype[0].charset = LATIN_CT;
dtype[0].size.length = strlen((const char *)regionName);
```

5. Call FNC\_CallSP.

## Example: Calling a Stored Procedure From an External Stored Procedure

Consider a stored procedure that has the following definition:

```
CREATE PROCEDURE addRegion (IN region VARCHAR(64),
                           OUT region_count INTEGER)
BEGIN
    INSERT INTO regionTable(:region);
    SELECT COUNT(*) INTO :region_count FROM regionTable;
END;
```

The following code calls the addRegion stored procedure from the external stored procedure xsp\_getregion:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xsp_getregion ( VARCHAR_LATIN *region,
                    int          *region_isnull,
                    char          sqlstate[6],
                    SQL_TEXT      extname[129],
                    SQL_TEXT      specific_name[129],
                    SQL_TEXT      error_message[257])
{
    char    tmp_string[64];
    void    *argv[2];
    int     ind[2];
    parm_t  dtype[2];
    INTEGER regionCount;    /* OUT argument from addRegion */

    /* Set the return indicator value for the external stored procedure*/
    *region_isnull = 0;

    if (strlen((const char *)region) > 4)
    {
        /* Strip off the first four characters */
        strcpy(tmp_string, (char *)region);
        strcpy((char *)region, &tmp_string[4]);
        /* Set the pointers to the stored procedure arguments */
        argv[0] = region;          /* IN */
    }
}
```

```

    argv[1] = &regionCount;    /* OUT */

    /* Set the indicator for the IN argument */
    ind[0] = 0;

    memset(dtype, 2, sizeof(parm_t)*2);

    /* Data type for the VARCHAR IN argument */
    dtype[0].datatype = VARCHAR_DT;
    dtype[0].direction = IN_PM;
    dtype[0].charset = LATIN_CT;
    dtype[0].size.length = strlen((const char *)region);

    /* Data type for the INTEGER OUT argument */
    dtype[1].datatype = INTEGER_DT;
    dtype[1].direction = OUT_PM;

    FNC_CallSP((SQL_TEXT *)"addRegion", 2, argv, ind, dtype, sqlstate);

    if (strcmp(sqlstate, "00000") != 0)
    {
        strcpy((char *)error_message, "Bad call to addRegion");
        return;
    }
}
else
{
    strcpy(sqlstate, "U0001");
    strcpy((char *)error_message, "Region string too short");
    return;
}
}

```

Here is the CREATE PROCEDURE statement that installs the xsp\_getregion external stored procedure on the server:

```

CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE TD_GENERAL;

```

## Nested Procedures

When doing nested stored procedure calls only one of those procedures can be an external procedure. For example, an external stored procedure cannot call a stored procedure that in turn calls an external stored procedure.

## Related Information

For more information on FNC\_CallSP, see [FNC\\_CallSP](#).

## Executing SQL in C/C++ External Stored Procedures

C or C++ external stored procedures can use CLlv2 to directly execute SQL.

## Overall Procedure

In general, a C or C++ external stored procedure that executes SQL follows standard CLI application programming practices. For more information on using CLlv2 to communicate with the database, see *Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems*, B035-2418.

Here are the basic steps for defining C or C++ external stored procedures that use CLlv2 to directly execute SQL. For complete code examples, see [External Stored Procedure Code Examples](#).

Step	Action	Code Example
1	Include the CLlv2 header files after the sqltypes_td.h header file.	<pre> #define SQL_TEXT Latin_Text #include &lt;sqltypes_td.h&gt; #include &lt;string.h&gt; #include &lt;stdio.h&gt; #include &lt;wchar.h&gt; #include &lt;coptypes.h&gt; #include &lt;coperr.h&gt; #include &lt;parcel.h&gt; #include &lt;dbcarea.h&gt;  ... </pre>
2	Declare the function, following the parameter passing convention you specify in the CREATE PROCEDURE or REPLACE PROCEDURE statement.	<pre> void ET001_xsp1( VARCHAR_LATIN *A_Name,                 VARCHAR_LATIN *B_Result,                 char            sqlstate[6]) {     ... } </pre>
3	Initialize the DBCAREA.	<pre> ...  DBCAREA dbcarea; Int32    result; </pre>



Step	Action	Code Example
		<pre> char    cntxt[4]; char    str1[200];  dbcarea.total_len = sizeof(struct DBCAREA);     DBCHINI(&amp;result, cntxt, &amp;dbcarea);  ... </pre>
4	<p>Establish a default connection by changing the Create Default Connection option to 'Y'.</p> <p>This step is different from standard CLI application programming.</p> <p>The default connection sends SQL to the session that the external stored procedure is running in.</p>	<pre> ...  dbcarea.change_opts = 'Y'; dbcarea.use_presence_bits = 'N'; dbcarea.keep_resp = 'N'; dbcarea.loc_mode = 'Y'; dbcarea.var_len_req = 'N'; dbcarea.save_resp_buf = 'N'; dbcarea.two_resp_bufs = 'N'; dbcarea.ret_time = 'N'; dbcarea.wait_for_resp = 'Y'; dbcarea.req_proc_opt = 'E'; dbcarea.req_buf_len = 1024; dbcarea.resp_buf_len = 1024; dbcarea.data_encryption = 'N';     dbcarea.create_default_connection = 'Y';     dbcarea.func = DBFCON;      DBCHCL(&amp;result, cntxt, &amp;dbcarea);  ... </pre>
5	<p>Initiate the SQL request, fetch the response, and end the request.</p> <p>(Repeat this step as needed.)</p>	<pre> ...  memset(str1, ' ', 100); sprintf(str1, "DELETE USER %s;", A_Name); dbcarea.func = DBFIRQ; dbcarea.req_ptr = str1; dbcarea.req_len = strlen(str1);     DBCHCL(&amp;result, cntxt, &amp;dbcarea); if (result != EM_OK) {     strcpy(sqlstate, "U0007");     return; }  ...  dbcarea.func = DBFFET;     DBCHCL(&amp;result, cntxt, &amp;dbcarea);  if (result != EM_OK) {     strcpy(sqlstate, "U0008"); } </pre>

Step	Action	Code Example
		<pre>         return;     }     ...      dbcarea.func = DBFERQ;     <b>DBCHCL(&amp;result, cntxt, &amp;dbcarea);</b>     if (result != EM_OK)     {         strcpy(sqlstate, "U0009");         return;     }     ... </pre>
6	Set any INOUT or OUT arguments. Perform cleanup, including releasing allocated memory and closing any connections to other systems. Return.	<pre>     ...      strcpy((char *)B_Result, "User deleted."); } </pre>

## Requesting an 8-byte Activity Count

If you want the database to return 8-byte activity counts to the SQL requests from the external stored procedure, both the parent session and the default session have to request for it. The 8-byte activity count is returned in the Enhanced Statement Status (ESS) response parcel; therefore, you can request for an 8-byte activity count by requesting for the ESS parcel.

A parent session that is running from BTEQ (or most of the other SQL processors) requests the ESS parcel by default. The external stored procedure code must also request for the ESS parcel and handle this parcel in the SQL responses from the database.

The following sample code shows how you can code your external stored procedure to request for the ESS parcel and handle the ESS parcel in the responses from the database. The relevant code is shown in bold font.

```

/* Set DBCAREA options. */

static void
set_options(DBCAREA *dbcarea_pt)
{
    dbcarea_pt->change_opts = 'Y';
    ...
    dbcarea_pt->data_encryption= 'N';
    dbcarea_pt->statement_status = 'E';
    dbcarea_pt->create_default_connection = 'Y';
}

```

```

}

/* Fetch parcels and check activity type. */

static Int16
fetch_request(DBCAREA *dbcarea_pt,
             int stmt_no,
             char *resultx,
             SQL_TEXT *error_message)
{
    ...
    struct CliSuccessType *SuccPcl;
    struct CliOkType *OKPcl;
    struct CliFailureType *FailPcl;
    struct CliStatementStatusType *EssPcl;
    ...
    while (status == OK)
    {
        DBCHCL(&result, cnta, dbcarea_pt);
        count=1;
        if (result == REQEXHAUST) status = STOP;
        else if (result != EM_OK) status = FAILED;
        else
        {
            switch(dbcarea_pt->fet_parcel_flavor)
            {
                case PclSUCCESS :
                    SuccPcl =
                        (struct CliSuccessType *)dbcarea_pt->fet_data_ptr;
                    memcpy(acc.acc_c, SuccPcl->ActivityCount, 4);
                    sprintf(strpstr, "[%d]Succ:Act(%d)|\0", stmt_no,
                        SuccPcl->ActivityType);
                    strcat(resultx, strpstr);
                    break;
                case PclOK :
                    OKPcl = (struct CliOkType *)dbcarea_pt->fet_data_ptr;
                    sprintf(strpstr, "[%d]OK:Act(%d)|\0", stmt_no,
                        OKPcl->ActivityType);
                    strcat(resultx, strpstr);
                    break;
                case PclSTATEMENTSTATUS:
                    EssPcl = (struct CliStatementStatusType *)
                        dbcarea_pt->fet_data_ptr;
                    sprintf(strpstr, "[%d]ESS:Act(%d)|\0", stmt_no,

```

```

        EssPcl->ActivityType);
    strcat(resultx, strptr);
    break;
case PclRECORD :
    memcpy(&item, dbcarea_pt->fet_data_ptr, 4);
    //cntx = item;
    break;
...
}

```

## Creating Result Sets to Return to the Client or Caller

If the CREATE PROCEDURE or REPLACE PROCEDURE statement for the external stored procedure specifies the DYNAMIC RESULT SETS clause, the external stored procedure can create dynamic result sets that are returned to the client application or to the caller of the external stored procedure upon completion of the external stored procedure.

A *result set* is a set of rows that is the result of a SELECT statement that the external stored procedure executes. An external stored procedure can create up to 15 result sets, depending on the specification of the DYNAMIC RESULT SETS clause.

To create a result set from an external stored procedure:

1. Follow the basic steps for defining C or C++ external stored procedures that use CLv2 to directly execute SQL, as described in [Overall Procedure](#).
2. Submit a single SELECT statement, setting the following options of the DBCAREA.

Option	Value	Description
Stored Procedure Return Result	2	Return the response to the SELECT statement to the client application.
	3	Return the response to the SELECT statement to the caller of the external stored procedure.
	4	Return the response to the SELECT statement to the client application and to the external stored procedure.
	5	Return the response to the SELECT statement to the caller of the external stored procedure and to the external stored procedure.
Keep Response	'Y'	The set of rows returned follows the rules for NO SCROLL cursors.
	'P'	The set of rows returned follows the rules for SCROLL cursors.

Here is a code excerpt for an external stored procedure that submits a SELECT statement and creates a result set to return to the caller of the external stored procedure:

```
...
```

```
memset(str1, ' ', 100);
sprintf(str1, "SELECT * FROM UserIds WHERE UName = %s;", A_Name);
dbcarea.func          = DBFIRQ;
dbcarea.req_ptr       = str1;
dbcarea.req_len       = strlen(str1);
dbcarea.change_opts   = 'Y';
dbcarea.SP_return_result = 3;
dbcarea.keep_resp      = 'Y';

DBCHCL(&result, cntxt, &dbcarea);

...
```

After the external stored procedure completes execution, the database returns the result sets created by the external stored procedure to the client or caller.

## Consuming Result Sets Created by Calling a Stored Procedure

An external stored procedure that uses CLlv2 to directly execute SQL can call a stored procedure that creates up to 15 dynamic result sets.

To indicate that the external stored procedure is willing to consume result sets created by a stored procedure, the SQL request that contains the CALL statement must set the Dynamic Result Sets Allowed option in the DBCAREA to 'Y'.

Here is a code excerpt for a request that calls a stored procedure called sp1:

```
...
dbcarea.change_opts          = 'Y';
dbcarea.dynamic_result_sets_allowed = 'Y';
dbcarea.req_ptr              = "CALL sp1('SEL * FROM t1');";
...
```

For a complete code example, see [Using CLlv2 to Consume Dynamic Result Sets](#). You can use the code example to see the types of parcels in a response that Vantage returns to the caller of a stored procedure that creates result sets.

## Calling Teradata Library Functions

Calling most of the Teradata library functions in an external stored procedure that uses CLlv2 involves several expensive context switches because the external stored procedure uses a different interface mode to execute most of the Teradata library functions.

Additionally, some of the Teradata library functions require that the external stored procedure wait for any outstanding CLlv2 requests to complete before calling them.

The exceptions are FNC\_malloc and FNC\_free, which do not incur any additional overhead.

The best practice for an external stored procedure that use CLIV2 to execute SQL is to use Teradata library functions judiciously.

For details on which library functions must wait for outstanding CLIV2 requests to complete before they are called, see [C Library Functions](#).

## Effects of Using DBFABT to Abort a Request

Using the DBFABT CLI function in an external stored procedure to abort a request causes the transaction (if any) to abort. The external stored procedure receives the 3110 user abort message and continues to run, without affecting the client application that called the external stored procedure.

## Returning Before an Outstanding Request Completes

An external stored procedure that returns from the C or C++ function before an outstanding request is complete generates a 7835 failure, aborting the transaction and the external stored procedure and any nested stored procedures if the external stored procedure was in a nested call. The failure is returned to the client application.

The best practice for an external stored procedure is to complete any outstanding request before returning from the function, clean up any logged on connections, and release allocated memory.

## Calling an External Stored Procedure From a Trigger

An external stored procedure that is called when a trigger is fired can only execute SQL statements that are allowed as triggered action statements.

Other statements result in the system returning a failure message to the external stored procedure. An external stored procedure that receives such a message has an opportunity to post the error and close any external files or disconnect any connections it established. The only remaining course of action is for it to return.

IF the external stored procedure receives a failure message and ...	THEN the triggering request is terminated and ...
returns with its own error by setting the SQLSTATE to its own exception code	the original fail condition will not be known to the caller of the external stored procedure.
returns with no error by setting SQLSTATE to '00000'	the system returns the original failure to the caller.
attempts to submit another request	the system generates a 7836 error: The XSP db.name submitted a request subsequent to receiving a trigger fail message.

For more details and a list of SQL statements that are allowed as triggered action statements, see the information about CREATE TRIGGER in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Related Information

FOR information on ...	SEE ...
the requirements that your system must meet before you can install or invoke an external stored procedure that uses CLlv2	<a href="#">System Requirements.</a>
code examples that uses CLlv2 to execute SQL	<a href="#">Using CLlv2 to Execute SQL.</a>
	<a href="#">Using CLlv2 to Consume Dynamic Result Sets.</a>
returning result sets from a stored procedure	<i>Teradata Vantage™ - SQL Stored Procedures and Embedded SQL</i> , B035-1148.
using CREATE PROCEDURE or REPLACE PROCEDURE to install an external stored procedure that uses CLlv2 on the server	<a href="#">Installing External Stored Procedures That Use CLlv2.</a>
using CLlv2	<i>Teradata® Call-Level Interface Version 2 Reference for Workstation-Attached Systems</i> , B035-2418.

## Installing a C/C++ External Stored Procedure

After you write and test a C or C++ external stored procedure, you can install it on the server.

### Note:

In general, you should not create UDFs, UDMs, or external stored procedures in Teradata system databases such as SYSLIB or SYSUDTLIB. These databases are primarily used for Teradata system UDFs, UDTs, UDMs, and external stored procedures only, and they usually contain a large number of these system external routines. Every time you create, alter, or drop your external routine in these databases, Teradata must relink your routine to all the objects of the system external routines. In addition, to execute your routine, Teradata must load all the shared libraries referenced by the system external routines, and these libraries may not be related to your routine. This is very inefficient. However, note that there are cases where you have to create your UDF in a system database. For example, UDFs used for row level security must reside in the SYSLIB database.

## CREATE PROCEDURE Statement

To identify the file name and location of the source code and install it on the server, use the form of the CREATE PROCEDURE statement for external stored procedures. You can submit the statement the same

way you submit the corresponding CREATE PROCEDURE for stored procedures, for example, by using the COMPILE command in BTEQ, but it is not required.

IF the external stored procedure ...	THEN the C/C++ function ...
uses CLIV2	is compiled, linked to a CLI-specific external stored procedure dynamic linked library (DLL or SO) associated with the database in which the external stored procedure resides, and distributed to all database nodes in the system.
does not use CLIV2	is compiled, linked to a standard dynamic linked library (DLL or SO) associated with the database in which the procedure resides, and distributed to all database nodes in the system. Note that other C/C++ external routines that are defined in the same database are also linked into the dynamically linked library.

## Default and Temporary Paths

To manage external stored procedures, Teradata uses default and temporary paths for external stored procedure creation and execution, including:

- A default directory that Teradata uses to search for source files and object files if the CREATE PROCEDURE does not explicitly specify the location.

If a source file or object file is on the server and the path is not fully specified in the CREATE PROCEDURE statement, the full path of the file is expected to begin here.

An administrator, or someone with sufficient privileges, is responsible for creating this directory.

- A temporary directory where external stored procedures are compiled.  
Any files needed for the compilation process are moved here. This includes source files from the server or client as well as object and header files, if needed.
- A directory where Teradata saves the dynamically linked libraries.
- A directory of shared memory files for external stored procedures that execute in protected mode.

For information, including the names of external stored procedure default and temporary paths, see *Teradata Vantage™ - Database Administration*, B035-1093.

## Specifying Source File Locations

The CREATE PROCEDURE statement provides clauses that specify the name and path of the function source code file.

CREATE PROCEDURE Clause	Description
EXTERNAL	Use the EXTERNAL clause when the function source is in the current or default directory on the client, and no other files need to be included.



CREATE PROCEDURE Clause	Description
	<p>The source name is the name that immediately follows the CREATE PROCEDURE keywords.</p> <p>If the client is..</p> <ul style="list-style-type: none"> <li>• workstation-attached, then BTEQ adds appropriate file extensions to the source name to locate the source file.</li> <li>• mainframe-attached, then the source name must be a DDNAME file name.</li> </ul> <p>Here is an example:</p> <pre>CREATE PROCEDURE GetRegionXSP   (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL PARAMETER STYLE TD_GENERAL;</pre>
EXTERNAL NAME ' <i>string</i> '	<p>Use '<i>string</i>' to specify names and locations of the following:</p> <ul style="list-style-type: none"> <li>• Function source, include header files, object files, libraries, and packages on the server.</li> <li>• Function source, include header files, and object files on the client.</li> </ul> <p>You can also use '<i>string</i>' to specify that the source or include files not be stored.</p> <p>If the client is..</p> <ul style="list-style-type: none"> <li>• workstation-attached, then if necessary, BTEQ adds appropriate file extensions to the names to locate the files.</li> <li>• mainframe-attached, then the names must be DDNAME file names.</li> </ul> <p>Here is an example:</p> <pre>CREATE PROCEDURE GetRegionXSP   (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion' PARAMETER STYLE TD_GENERAL;</pre> <p>where...</p> <ul style="list-style-type: none"> <li>• ! in '<i>string</i>' specifies a delimiter.</li> <li>• C in '<i>string</i>' specifies that the source is obtained from the client.</li> <li>• S in '<i>string</i>' specifies that the information between the following two sets of delimiters identifies the name and location of a C or C++ function source file.</li> <li>• getregion in '<i>string</i>' specifies the name, without the file extension, that the server uses to compile the source.</li> <li>• xspsrc/getregion.c in '<i>string</i>' specifies a relative path (xspsrc) for the source file (getregion.c).</li> <li>• F in '<i>string</i>' specifies that the information after the next delimiter identifies the C or C++ function name.</li> <li>• xsp_getregion in '<i>string</i>' specifies the C or C++ function name.</li> </ul>

CREATE PROCEDURE Clause	Description
EXTERNAL NAME <i>function_name</i>	<p>Use EXTERNAL NAME <i>function_name</i> when the function source is in the current or default directory on the client, and no other files need to be included.</p> <p>The source name is the same as <i>function_name</i>.</p> <p>If the client is...</p> <ul style="list-style-type: none"> <li>workstation-attached, then BTEQ adds appropriate file extensions to <i>function_name</i> to locate the source file.</li> <li>mainframe-attached, then <i>function_name</i> must be a DDNAME file name.</li> </ul> <p>Here is an example where <i>function_name</i> is xsp_getregion:</p> <pre>CREATE PROCEDURE GetRegionXSP   (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL NAME xsp_getregion PARAMETER STYLE TD_GENERAL;</pre>

For more information on CREATE PROCEDURE and the EXTERNAL clause, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Source File Locations for ODBC

If you use ODBC, you can only create external stored procedures from files that are stored on the server.

## Source File Locations for JDBC

Using Teradata Driver for the JDBC Interface, you can create external stored procedures from files that are located on the Teradata server, or from resources located on the client.

A client-side external stored procedure source file must be available as a resource in the class path. The Teradata JDBC driver can load the resource from the class path and transfer it to the server node without directly accessing the client file system.

## Specifying the C/C++ Function Name

The CREATE PROCEDURE statement provides clauses that identify the C/C++ function name that appears immediately before the left parenthesis in the C/C++ function declaration or the function entry name when the C/C++ object is provided instead of the C/C++ source.

IF CREATE PROCEDURE specifies ...	THEN ...
EXTERNAL	The C/C++ function name must match the name that follows the CREATE PROCEDURE keywords.

IF CREATE PROCEDURE specifies ...	THEN ...
	<p>Consider the following CREATE PROCEDURE statement:</p> <pre>CREATE PROCEDURE GetRegionXSP   (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL PARAMETER STYLE TD_GENERAL;</pre> <p>The C function name must be GetRegionXSP. If the client is mainframe-attached, then the C/C++ function name must be the DDNAME for the source.</p>
EXTERNAL NAME <i>function_name</i>	<p>the C/C++ function name must match <i>function_name</i>. Consider the following CREATE PROCEDURE statement:</p> <pre>CREATE PROCEDURE GetRegionXSP   (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL NAME xsp_getregion PARAMETER STYLE TD_GENERAL;</pre> <p>The C function name must be xsp_getregion. If the client is mainframe-attached, then <i>function_name</i> must be the DDNAME for the source.</p>
EXTERNAL NAME ' <i>string</i> '	<p>'<i>string</i>' can include the F option to specify the C/C++ function name. Consider the following CREATE PROCEDURE statement:</p> <pre>CREATE PROCEDURE GetRegionXSP   (INOUT region VARCHAR(64)) LANGUAGE C NO SQL EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion' PARAMETER STYLE TD_GENERAL;</pre> <p>The C function name must be xsp_getregion. If '<i>string</i>' does not include the F option, then the C/C++ function name must match the name that follows the CREATE PROCEDURE keywords.</p>

## Specifying Nonstandard Include Files

If an external stored procedure includes a nonstandard header file, the EXTERNAL clause in the CREATE PROCEDURE statement must specify the name and path of the header file.

Consider the following external stored procedure that includes the header file *rtypes.h*:

```

/***** C source file name: get_r.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include "rtypes.h"

void xsp_getr( VARCHAR_LATIN *region,
              char          sqlstate[6])
{
    ...
}

```

Here is an example of CREATE PROCEDURE that specifies the name and path of the nonstandard header file:

```

CREATE PROCEDURE GetRegionXSP (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME
'CI!rtypes!xsp_home/rtypes.h!CS!get_r!xsp_home/get_r.c!F!xsp_getr'
PARAMETER STYLE TD_GENERAL;

```

where:

This part of the string that follows EXTERNAL NAME ...	Specifies ...
!	a delimiter.
C	that the header file is obtained from the client.
I	that the information between the following two sets of delimiters identifies the name and location of an include file (.h).
rtypes	the name, without the file extension, of the header file.
xsp_home/rtypes.h	the path and name of the header file on the client.
C	that the source is obtained from the client.
S	that the information between the following two sets of delimiters identifies the name and location of a C or C++ function source file.
get_r	the name, without the file extension, that the server uses to compile the source.
xsp_home/getregion.c	the path and name of the source file.

This part of the string that follows <b>EXTERNAL NAME ...</b>	Specifies ...
F	that the information after the next delimiter identifies the C or C++ function name.
xsp_getregion	the C or C++ function name.

For more information on installing libraries, see the information about CREATE PROCEDURE in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Installing External Stored Procedures That Use CLlv2

The CREATE PROCEDURE or REPLACE PROCEDURE statement that you use to install an external stored procedure that uses CLlv2 must specify the following options:

- An EXTERNAL NAME clause that includes the Teradata CLI package name
- One of the following options that identify whether the SQL statements that the external stored procedure executes read or modify SQL data in the database:
  - CONTAINS SQL
  - READS SQL DATA
  - MODIFIES SQL DATA

Here is an example

```
REPLACE PROCEDURE ET001_xsp1(IN A_Name VARCHAR(10),
                             OUT resultx VARCHAR(16000))
  LANGUAGE C
  MODIFIES SQL DATA
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'SP!CLI!CS!ET001_xsp1!ET001_xsp.c';
```

If the external stored procedure returns dynamic result sets to the client application or the caller of the external stored procedure, the CREATE PROCEDURE or REPLACE PROCEDURE statement must also specify the DYNAMIC RESULT SETS option.

Here is an example:

```
REPLACE PROCEDURE ET001_xsp1(IN A_Name VARCHAR(10),
                             OUT resultx VARCHAR(16000))
  LANGUAGE C
  READS SQL DATA
  PARAMETER STYLE TD_GENERAL
  DYNAMIC RESULT SETS 2
  EXTERNAL NAME 'SP!CLI!CS!ET001_xsp1!ET001_xsp.c';
```

## Related Information

FOR more information on ...	SEE ...
the CREATE PROCEDURE statement	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the privileges that apply to external stored procedures	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.
Linux limitations that affect name and number of external routines in a database	<a href="#">Argument list too long</a> .

## Debugging an External Stored Procedure

When debugging an external stored procedure, be sure your tests include the following:

- Limit checks on input values
- Limit checks on return values
- Proper handling of NULLs
- Division by zero
- All memory acquired by *malloc* is freed
- All open operating system handles are released/closed

The Teradata C/C++ UDF Debugger allows external stored procedures to be debugged within the database on a development or test system. The debugger only supports C and C++ external stored procedures. For more information on the Teradata C/C++ UDF Debugger, see [C/C++ Command-line Debugging for UDFs](#).

For other external stored procedures, debugging is limited after they are installed in the database. You can use other debugging tools to verify their functionality, or run them on a standalone virtual machine and debug the UDF server processes that executes them.

## Forcing an SQL Warning Condition

You can force an SQL warning condition at the point in the code where the external stored procedure appears to have problems. To force the warning, set the *sqlstate* return argument to '01H xx', where you choose the numeric value of xx.

If you use parameter style SQL, you can also set the *error\_message* return argument to return up to 256 SQL\_TEXT characters.

The warning is issued as a return state of the procedure. The warning does not terminate the transaction; therefore, you must set return arguments to valid values that the transaction can use.

Only one warning can be returned per invocation.

You force an SQL warning condition in a stored procedure in the same manner you force an SQL warning condition in a UDF. For an example of how to set the SQLSTATE result code and error message return argument in a UDF, see [Forcing an SQL Warning Condition](#).

## Using Trace Tables

You can use trace tables to get trace diagnostic output.

External stored procedures use trace tables in the same manner as UDFs use trace tables. For a description of how to debug UDFs using trace tables, see [Debugging Using Trace Tables](#).

## Resolving UDF Server Setup Errors

When an external routine like an UDF, UDM, or external stored procedure is called, a UDF server process is acquired from the UDF server pool to execute the external routine. If there are no UDF server processes in the pool or if all of the processes in the pool are busy, then the system tries to start a new UDF server process for the request.

The startup of the new UDF server usually takes some time, especially if the UDF server is for executing Java external routines, or if the system is very busy. If the new UDF server cannot be started within the default time limit, the query that contains the UDF, UDM, or external procedure call is aborted, and you may receive a 7583 error indicating that the UDF server setup encountered a problem. The system log may also record a 7820 error specifying that the UDF server could not stay up long enough for initialization.

If you are experiencing these errors, you can contact Teradata Support Center personnel to adjust the time limit allowed for starting a new UDF server process. For details, see the information about the Cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## External Stored Procedure Invocation

Invoking an external stored procedure in a CALL statement is no different from invoking a stored procedure.

## Argument List

The arguments in the CALL statement must appear as comma-separated expressions in the same order as the C or C++ function declaration parameters.

The argument types of an external stored procedure in a CALL statement must either be compatible with the corresponding parameter declarations in the function declaration or must be types that are implicitly converted to the corresponding parameter types, according to the database implicit type conversion rules.

To pass an argument that is not compatible with the corresponding parameter type and is not implicitly converted by the database, the CALL statement must explicitly convert the argument to the proper type.

For information on data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Invoking External Stored Procedures with TD\_ANYTYPE OUT Parameters

When invoking an external stored procedure that is defined with a TD\_ANYTYPE OUT parameter, you can specify the RETURNS *data type* or RETURNS STYLE *column expression* clauses along with the OUT argument in the CALL statement to indicate the desired return type of the OUT parameter. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

For detailed information, see [External Stored Procedures That Use TD\\_ANYTYPE Arguments](#).

## Nested Procedure Calls

When doing nested stored procedure calls only one of those procedures can be an external procedure.

## Related Information

For more information on the CALL statement, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## Protected Mode Execution

### Protected Mode Execution Option

The ALTER PROCEDURE statement provides an execution option that controls whether the database invokes the external stored procedure directly or runs the procedure indirectly as a separate process.

The option applies to external stored procedures that do not execute SQL and were created without specifying the EXTERNAL SECURITY clause in the CREATE PROCEDURE or REPLACE PROCEDURE statement. External stored procedures that specify the EXTERNAL SECURITY clause are executed using separate secure server processes.

IF ALTER PROCEDURE specifies ...	THEN Vantage ...
EXECUTE PROTECTED	runs the external stored procedure indirectly as a separate process. If the stored procedure fails during execution, the transaction fails.
EXECUTE NOT PROTECTED	invokes the external stored procedure directly.

### NOTICE

If the ALTER PROCEDURE statement specifies EXECUTE NOT PROTECTED, and the external stored procedure fails during execution, the database software will probably restart.



Only an administrator, or someone with sufficient privileges, can use the ALTER PROCEDURE statement.

## Choosing the Correct Execution Option

Use the following table to choose the correct execution option for external stored procedures that do not execute SQL and were created without specifying the EXTERNAL SECURITY clause in the CREATE PROCEDURE or REPLACE PROCEDURE statement.

IF ...	THEN use ...
you are in the development phase and are debugging an external stored procedure	EXECUTE PROTECTED.
the external stored procedure does not use any operating system resources	EXECUTE NOT PROTECTED. Running an external stored procedure in nonprotected mode speeds up processing considerably. Use this option only after thoroughly debugging the external stored procedure and making sure it produces the correct output.

## Related Information

FOR more information on ...	SEE ...
protected mode process and server administration	<a href="#">Protected Mode Process and Server Administration for C/C++ External Routines.</a>
privileges associated with external stored procedures	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.
ALTER PROCEDURE and the EXECUTE PROTECTED option	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
CREATE PROCEDURE and the 'D' character in the EXTERNAL NAME 'string' clause	

## AT TIME ZONE Option for External Procedures

When you create an external procedure, the database stores the current session time zone for the procedure along with its definition to enable the SQL language elements in the procedure to execute in a consistent time zone and produce consistent results. However, time or timestamp data passed as an input parameter to the procedure still use the runtime session time zone rather than the creation time zone for the procedure.

The AT TIME ZONE option of the ALTER PROCEDURE statement enables you to reset the time zone for all of the SQL elements of external procedures when you recompile a procedure. The database then stores the newly specified time zone as the creation time zone for the procedure.

You can only specify AT TIME ZONE with the COMPILE [ONLY] option, and it must follow the COMPILE [ONLY] specification. If it does not, the database aborts the request and returns an error to the requestor. For details, see the information about ALTER PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Argument Behavior

FOR information on ...	SEE ...
behavior when arguments are null	<a href="#">Behavior When Using NULL as a Literal Argument.</a>
truncation of character string arguments	<a href="#">Truncation of Character String Arguments.</a>
overflow and numeric arguments	<a href="#">Overflow and Numeric Arguments.</a>
overflow error related to value of data type	<a href="#">Data Type for INOUT Constant Arguments.</a>

## Truncation of Character String Arguments

The session transaction mode affects character string truncation.

IF the session transaction mode is ...	THEN an input character string that requires truncation is truncated ...
Teradata	without reporting an error. Truncation on Kanji1 character strings containing multibyte characters might result in truncation of one byte of the multibyte character.
ANSI	of excess pad characters without reporting an error. Truncation of other characters results in a truncation exception.

The normal truncation rules apply to a result string. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Behavior When Using NULL as a Literal Argument

IF an input argument is the NULL keyword and the parameter style is ...	THEN ...
SQL	the external stored procedure is called with the appropriate indicators set to the null indication.
TD_GENERAL	an error is reported.

## Overflow and Numeric Arguments

To avoid numeric overflow conditions, the C/C++ function should define a decimal data type as big as it can handle.

If the assignment of the value of an input or output numeric argument would result in a loss of significant digits, a numeric overflow error is reported.

For example, consider an external stored procedure that takes a DECIMAL(2,0) argument:

```
CREATE PROCEDURE smldec( IN p1 DECIMAL(2,0) )
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL;
```

Passing a number with a maximum of two digits is successful:

```
CALL smldec(99);
```

An attempt to pass a number larger than 99 or smaller than -99 would result in a loss of significant digits.

```
CALL smdec(100);
```

```
Failure 2616 Numeric overflow occurred during computation.
```

Any fractional numeric data that is passed or returned that does not fit as it is being assigned is rounded according to the Teradata rounding rules. For more information on rounding, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Data Type for INOUT Constant Arguments

The data type for an INOUT constant argument is governed by the data type of the value passed in, not what is defined. If the data type of the value passed in is smaller than the data type defined in the CREATE PROCEDURE or REPLACE PROCEDURE statement, and the external stored procedure returns a value larger than the maximum value of the data type for the value passed in, the system returns an overflow error.

For example, consider an external stored procedure that defines an INTEGER INOUT parameter:

```
CREATE PROCEDURE inout_example( INOUT p1 INTEGER )
LANGUAGE C
NO SQL
```

```
PARAMETER STYLE SQL  
EXTERNAL;
```

If you call the external stored procedure with a constant input value that fits into a SMALLINT, the system returns an overflow error if the output value is larger than 32767, the maximum value of a SMALLINT:

```
CALL inout_example(1000);
```

## C/C++ User-Defined Methods

A user-defined method (UDM) is a special kind of UDF that is associated with a user-defined type (UDT).

You write UDMs in the C or C++ programming language, install them in the database, and then use the `udt_expression.method_name` dot notation to invoke them.

This section uses the term *method* and the acronym UDM interchangeably. The term *type* and the acronym UDT are also interchangeable.

### UDM Types

Teradata supports two types of UDMs:

- Instance
- Constructor

#### Instance Methods

An instance method operates on a specific instance of a distinct or structured UDT. For example, an instance method named *area* might calculate and return the area of a structured UDT named *circle* that contains attributes *x*, *y*, and *radius*.

You can replace a column name in an SQL expression with an instance method invocation on a UDT. When Vantage evaluates the expression, it invokes the UDM.

Instance methods can also provide transform, ordering, and cast functionality for a distinct or structured UDT. You do not invoke these types of instance methods directly. Vantage uses this functionality during certain operations involving the UDT.

WHEN you ...	THEN Vantage uses this functionality ...
convert the UDT to a predefined data type or another UDT, for example with the CAST function	cast
convert another UDT or predefined data type to the UDT	
compare two UDTs, for example with the ORDER BY clause	ordering
export the UDT from the server	transform Vantage also uses transform functionality to import the UDT to the server, for example with the USING modifier. However, the implementation is always a UDF, not a UDM.

Before you can create a table that has a UDT column and perform queries on the column, the UDT must have UDFs or instance methods associated with it that provide transform, ordering, and cast functionality.

IF the UDT is ...	THEN ...
distinct	Vantage automatically generates UDFs and methods that provide transform, ordering, and cast functionality.
structured	you are responsible for implementing UDFs or UDMs that provide cast, ordering, and transform functionality.

## Constructor Methods

A constructor method initializes an instance of a structured UDT.

A structured UDT can have more than one constructor method, each one providing different initialization options.

To create an instance of a structured UDT in an SQL statement, you can invoke the constructor method or specify the constructor method in a NEW expression.

## Overall Development Synopsis

### Procedure

Here is a synopsis of the steps you take to develop, compile, install, and use a UDM:

1. Write, test, and debug the C or C++ code for the UDM.

You can use the Teradata C/C++ UDF Debugger, which is a version of GDB (the gnu Source-Level Debugger) that contains extensions for the database. For more information, see [C/C++ Command-line Debugging for UDFs](#).

2. Use CREATE TYPE to create the UDT and specify constructor methods and instance methods.

IF the method is ...	THEN use this METHOD specification in the CREATE TYPE statement ...
an instance method	INSTANCE METHOD or METHOD
a constructor method for a structured type	CONSTRUCTOR METHOD

3. Use CREATE METHOD or REPLACE METHOD to identify the location of the source code or object, and install it on the server.

#### Note:

In general, you should not create UDMs in Teradata system databases such as SYSLIB or SYSUDTLIB. For more information, see [Installing the UDM](#).

The method is compiled, if the source code is submitted, linked to the dynamic linked library (DLL or SO) associated with the database in which the method resides, and distributed to all database nodes in the system.

- If the UDM is an instance method that provides transform, ordering, or cast functionality for a structured UDT, register the UDM as a transform, ordering, or cast routine. Otherwise, skip to the next step.

IF the UDM implements this functionality for a UDT ...	THEN use this statement to register the UDM ...
cast	CREATE CAST or REPLACE CAST
ordering	CREATE ORDERING or REPLACE ORDERING
transform	CREATE TRANSFORM or REPLACE TRANSFORM

- Test the UDM in *protected* execution mode until you are satisfied it works correctly.

Protected mode is the default execution mode for a UDM. In protected mode, the database isolates all of the data the UDM might access as a separate process in its own local workspace. This makes the method run slower. If any memory violation or other system error occurs, the error is localized to the method and the transaction executing the method.

- Use ALTER METHOD to change the UDM to run in nonprotected execution mode.
- Rerun the tests from Step 5 to test the UDM in nonprotected execution mode until you are satisfied it works correctly.
- Use GRANT to grant privileges to users who are authorized to use the UDT.

## Related Information

FOR more information on ...	SEE ...
developing a UDM	related topics in this document.
debugging a UDM	<a href="#">Debugging a UDM.</a>
code examples for UDMs	<a href="#">UDM Code Examples.</a>
<ul style="list-style-type: none"> <li>• CREATE CAST</li> <li>• CREATE METHOD</li> <li>• CREATE ORDERING</li> <li>• CREATE TRANSFORM</li> <li>• CREATE TYPE</li> <li>• REPLACE CAST</li> <li>• REPLACE METHOD</li> <li>• REPLACE ORDERING</li> <li>• REPLACE TRANSFORM</li> </ul>	<ul style="list-style-type: none"> <li>• related topics in this document.</li> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> </ul>

## UDM Source Code Development

You develop the body of a UDM using the C or C++ programming language.

### Note:

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

### Source Code Contents

The topics that follow provide details about the contents of the C or C++ source code for a UDM. In general, the contents of the C or C++ source code must:

- Define the SQL\_TEXT constant
- Include the sqltypes\_td.h header file
- Define the parameter list in the order that matches the parameter passing convention specified in the METHOD specification in the CREATE TYPE statement

### Internal and External Data

All data is local to the UDM. No global data can be retained from method invocation to method invocation, with the exception of GLOP data. For details on using GLOP data, see [Global and Persistent Data](#).

The UDM cannot contain any static variables.

A UDM can contain static constants.

### I/O

The following rules apply to I/O.

IF the UDM ...	THEN the UDM ...
runs in nonprotected execution mode	cannot do any I/O, including standard I/O and network I/O.
runs in protected execution mode	can do I/O or otherwise interface with the operating system such that the operating system retains resources such as file handles or object handles. The method executes as a separate process under the authorization of the 'tdatuser' operating system user. The method must release opened resources (close handles) before it completes. For more information on protected execution mode, see <a href="#">Protected Mode Execution</a> .
specifies the EXTERNAL SECURITY clause in the CREATE METHOD or REPLACE METHOD statement	can do I/O or otherwise interface with the operating system such that the operating system retains resources such as file handles or object handles. The method executes as a separate process under the authorization of a specific native operating system user established by a CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement. The method must release opened resources (close handles) before it completes.



## Signal Handling

UDMs that run on Linux systems cannot use or modify the handling of the signal SIGUSR2, which is reserved by Vantage.

## Standard C Library Functions

A UDM that runs in nonprotected mode or protected mode can call standard C library functions that do not do any I/O, such as string library functions.

Restrictions apply to some functions, such as *malloc* and *free*.

For more information, see [Using Standard C Library Functions](#).

## Teradata Library Functions

Teradata provides functions that you can use for UDM development. For example, you can use the FNC\_GetDistinctValue function to get the value of the distinct type that a UDM is associated with.

For more information, see [C Library Functions](#).

## Problems Using System V IPC and POSIX IPC

Teradata recommends that UDFs, UDMs, and external stored procedures not use System V IPC or POSIX IPC such as semaphore, mutex, conditional variable, and shared memory. The Teradata C library does not provide FNC functions to allocate and deallocate these IPC resources. When a session aborts or unexpected events cause Teradata to terminate the external routine execution threads or processes, there is a risk that these IPC resources may be left over in the system without being cleaned up. In rare cases, this may cause a system hang or OS resource leak.

## Consequences of Using the exit() System Call

Avoid calling exit() from UDMs.

UDMs that call exit() generate the following results:

- The request that called the UDM is rolled back.
- The server process is terminated and must be recreated for the next UDM, causing additional overhead.

## Header Files

Teradata provides the `sqltypes_td.h` header file that you include in your source code file. The header file defines the equivalent C data types for all database data types that you can use for the input arguments and result of your UDMs. Every SQL data type corresponds to a C data type in `sqltypes_td.h`.

## Location

The header file is in the `etc` directory of the Teradata software distribution:

```
/usr/tdbms/etc
```

To verify the path of the `etc` directory, enter the following on the command line:

pdepath -e

### SQL\_TEXT Definition

Before you include the `sqltypes_td.h` header file, you must define the `SQL_TEXT` constant. The value that you use must match the current server character set of the session in which you use the `CREATE METHOD` statement to create the UDM.

IF you use the <code>CREATE METHOD</code> statement when the current server character set is ...	THEN the C or C++ function must set <code>SQL_TEXT</code> to ...
KANJI	Kanji1_Text
KANJISJIS	Kanjisjis_Text
LATIN	Latin_Text
UNICODE	Unicode_Text

`SQL_TEXT` is used for specific UDM arguments when the UDM uses parameter style SQL. For details on parameter styles, see [UDM Parameter List](#).

Vantage remembers the character set the UDM was created under so that the UDM can translate `SQL_TEXT` input arguments to the expected text and translate text to `SQL_TEXT` output arguments, no matter who invokes the UDM and what the current server character set is for the session.

### NOTICE

In accordance with Teradata internationalization plans, KANJI1 support is deprecated and is to be discontinued in the near future. KANJI1 is not allowed as a default character set; the system changes the KANJI1 default character set to the UNICODE character set. Creation of new KANJI1 objects is highly restricted. Although many KANJI1 queries and applications may continue to operate, sites using KANJI1 should convert to another character set as soon as possible.

### Example: Defining `SQL_TEXT` and Including `sqltypes_td.h`

The following example shows how to define `SQL_TEXT` and include the `sqltypes_td.h` header file in the file that defines a UDM:

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
```

Alternatively, you can use the following line to include the `sqltypes_td.h` header file:

```
#include <sqltypes_td.h>
```

Using angle brackets (`<>`) or double quotation marks (`" "`) affects the path that the C preprocessor uses to search for the `sqltypes_td.h` header file.

## SQL Data Types

Every SQL data type corresponds to a C data type that you use for the input arguments and result of your UDM.

For more information, see [SQL Data Type Mapping](#).

For exact definitions, see the `sqltypes_td.h` header file.

## C/C++ Function Name

The name of the method in the C or C++ source code follows the C function naming conventions, with the additional restriction that the name cannot be longer than 30 characters.

In C++, the function definition must be preceded by `extern "C"` so that the function name is not converted to a C++ overloaded name. For example:

```
extern "C"
void get_area( UDT_HANDLE *circleUDT, FLOAT *result, char sqlstate[6])
{
    ...
}
```

The function can call any module written in C++.

When you use the `CREATE METHOD` statement to install the UDM, you specify the name of the C or C++ function. For more information, see [Installing the UDM](#).

On Linux, long names can sometimes cause errors when you install a UDM. For more information, see [Argument list too long](#).

## UDM Parameters

### Parameter List

The list of parameters for a UDM is very specific. It includes the following:

- The handle of the UDT with which the UDM is associated
- Input parameters that are specified during method invocation
- Output parameters that return the result of the UDM and the `SQLSTATE` result code

A UDM can have 0 to 128 input parameters. The types that you use as input parameters correspond to SQL data types in the method invocation. For more information, see [SQL Data Type Mapping](#).

Additional parameters might be required, depending on the parameter passing convention you choose.

Parameters are almost always specified as pointers to the data.

## Compatible Types

The argument types passed in to a UDM during method invocation do not always have to be an exact match with the corresponding parameter declarations in the function definition, but they must be *compatible* and follow the precedence rules that apply to compatible types.

For more information, see [Compatible Types](#).

## Parameter Passing Convention

UDMs support two types of parameter passing conventions.

Parameter Passing Convention	Description
Parameter Style SQL	Provides a way to pass nulls as input arguments and return a null as the result.
Parameter Style TD_GENERAL	Does not accept null input arguments and does not return a null result.

The parameter passing convention you use to code a UDM must correspond to the parameter passing specification of the METHOD specification in the CREATE TYPE statement for the UDT.

IF the METHOD specification in the CREATE TYPE statement specifies ...	THEN use the following syntax for the function parameter list ...
PARAMETER STYLE SQL or omits the PARAMETER STYLE option	<a href="#">Syntax for Parameter Style SQL</a>
PARAMETER STYLE TD_GENERAL	<a href="#">Syntax for Parameter Style TD_GENERAL</a>

## Method Name Overloading

UDMs support method name overloading in the same manner that UDFs support function name overloading: you can define several methods that have the same name but are different from each other in such a way to make each method unique.

The name that is overloaded is specified in the METHOD specification in the CREATE TYPE statement as the name to use to invoke the UDM. You must provide a unique C or C++ function for each UDM with the same name.

### Characteristics of a Unique Method

Methods that have the same name are unique if any of the following is true:

- The number of parameters is different.
- The parameter data types are distinct from each other.

For the rules on which parameter data types are distinct from each other, see [Parameter Types in Overloaded Functions](#).

## Relationship to CREATE TYPE and CREATE METHOD Statements

The name that is overloaded is the name that immediately follows the METHOD keyword in the CREATE TYPE statement.

Each time you use the METHOD specification in the CREATE TYPE statement to overload a method name, you must specify:

- A parameter list that satisfies the characteristics of a unique function
- A unique name in the SPECIFIC clause

Each time you use the CREATE METHOD statement to install an overloaded method, you must specify a different C or C++ function.

## Calling a Method That is Overloaded

The rules that Vantage uses to determine which method to invoke when the method is overloaded are the same rules used to determine which function to invoke when a user calls a UDF that is overloaded.

For details, see [Calling a Function That is Overloaded](#).

## Using TD\_ANYTYPE Parameters as an Alternative to Overloading Method Names

You can use TD\_ANYTYPE parameters to reduce the number of overloaded methods needed to support routines that have constant parameter counts but differing parameter data types. TD\_ANYTYPE parameters can accept any system-defined data type or user-defined type; therefore, you can create a single method that can support a multitude of data types.

For more information, see [Using TD\\_ANYTYPE Parameters as an Alternative to Overloading Function Names](#).

## UDM Parameter List

The first parameter of the C/C++ function for a UDM is always the handle of the UDT on which the UDM is invoked. The UDM uses the UDT handle to get the values of the UDT on which it is invoked.

The parameters that follow the UDT handle parameter correspond to the parameter list in the METHOD specification of the CREATE TYPE statement.

Use parameter style SQL to allow a method to pass nulls for arguments; otherwise, use parameter style TD\_GENERAL.

When you use parameter style SQL, you must define indicator parameters for each parameter.

## UDM Parameter List TD\_GENERAL Syntax

```
void method_name (
    UDT_HANDLE *udthandle,
    [ input_parameter_specification [...] ]
    result_type *result,
    char sqlstate[6]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

## UDM Parameter List SQL Syntax

```
void method_name (
    UDT_HANDLE *udthandle,
    [ input_parameter_specification [...] ]
    result_type *result,
    [ indicator_parameter_specification [...] ]
    int indicator_result,
    char sqlstate[6]
    SQLTEXT method_name [m]
    SQLTEXT specific_method_name [l]
    SQLTEXT error_message [p]
)
{
    ...
}
```

### *input\_parameter\_specification*

```
type *input_parameter,
```

***indicator\_parameter\_specification***

```
int *indicator_parameter,
```

**UDM Parameter List Syntax Elements*****method\_name***

Pointer to a C string whose value is the method name in the CREATE TYPE definition.

The UDT can use this name to build error messages.

***udthandle***

Handle of the UDT on which the UDM is invoked, also called the *subject parameter*.

***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE TYPE definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is the C type in `sqltypes_td.h` that corresponds to the SQL data type of *input\_parameter*.

***result***

Pointer to a data area big enough to hold the result that the UDT returns, as defined by the RETURNS clause in the corresponding CREATE TYPE statement.

***indicator\_parameter\_specification***

[Optional] Indicator parameter corresponding to an input parameter.

Each *input\_parameter\_specification* must have a corresponding *indicator\_parameter\_specification*. The input parameters and indicator parameters must be in the same order.

If the value of *indicator\_parameter* is -1, the value of the corresponding *input\_parameter* is null.

If the value of *indicator\_parameter* is 0, the value of the corresponding *input\_parameter* is a non-null value.

***indicator\_result***

Result indicator parameter corresponding to the result.

***sqlstate***

Pointer to a six-character C string that indicates the SQLSTATE value—success, exception, or warning. The first five characters are ASCII and the sixth is the C null character. The string is initialized to '00000', which indicates success.

For more information on SQLSTATE values, see [Returning SQLSTATE Values](#).

***m***

Number of characters in the method name in the CREATE TYPE definition. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a method name.

***specific\_method\_name***

Pointer to a C string whose value is the name of the external method being invoked.

If the CREATE TYPE statement includes the SPECIFIC clause, *specific\_method\_name* is the name in the SPECIFIC clause; otherwise, *specific\_method\_name* is the same as *method\_name*.

The UDT can use this name to build error messages.

***l***

Number of characters in the name of the external method. The ANSI SQL standard defines the maximum value for *m* as 128. Vantage allows a maximum of 30 characters for a method name.

***error\_message***

Pointer to a C string whose value is the the error message text.

***p***

Number of characters in the error message text. The maximum value is 256.

**Related Information:**

[Argument Behavior](#)

## Examples

**Example: UDM with Parameter Style TD\_GENERAL**

Here is a code excerpt that shows how to declare a C function for a UDM that uses parameter style TD\_GENERAL:



```

/***** C source file name: to_inches.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void meters_toInches( UDT_HANDLE *meterUDT
                     FLOAT      *result,
                     char        sqlstate[6])
{
    ...
}

```

For a complete example of the C function, see [UDM Code Examples](#).

The CREATE TYPE statement to create the UDT with which the method is associated looks like this:

```

CREATE TYPE meter AS FLOAT
FINAL
    INSTANCE METHOD toInches()
    RETURNS FLOAT
    SPECIFIC toInches
    NO SQL
    PARAMETER STYLE TD_GENERAL
    DETERMINISTIC
    LANGUAGE C;

```

The corresponding CREATE METHOD statement to install the UDM on the server looks like this:

```

CREATE METHOD toInches()
RETURNS FLOAT
FOR meter
EXTERNAL NAME 'CS!toinches!udm_src/to_inches.c!F!meters_toInches';

```

### Example: UDM with Parameter Style SQL

Here is an example of how to declare a C function for a UDM that uses parameter style SQL:

```

/***** C source file name: to_inches.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

```

```

void meters_toInches( UDT_HANDLE  *meterUDT
                     FLOAT        *result,
                     int           *meterUDTIsNull,
                     int           *resultIsNull,
                     char          sqlstate[6])
                     SQL_TEXT      extname[129],
                     SQL_TEXT      specific_name[129],
                     SQL_TEXT      error_message[257] )
{
    ...
}

```

For a complete example of the C function, see [UDM Code Examples](#).

The CREATE TYPE statement to create the UDT with which the method is associated looks like this:

```

CREATE TYPE meter AS FLOAT
FINAL
    INSTANCE METHOD toInches()
    RETURNS FLOAT
    SPECIFIC toInches
    NO SQL
    PARAMETER STYLE SQL
    DETERMINISTIC
    LANGUAGE C;

```

The corresponding CREATE METHOD statement to install the UDM on the server looks like this:

```

CREATE METHOD toInches()
RETURNS FLOAT
FOR meter
EXTERNAL NAME 'CS!toinches!udm_src/to_inches.c!F!meters_toInches';

```

## C/C++ Function Body

### Basic C/C++ Function Definition

Here are the basic steps you take to define a C/C++ function as a UDM:

1. Define the SQL\_TEXT constant.  
For more information, see [Header Files](#).
2. Include the sqltypes\_td.h header file.  
For more information, see [Header Files](#).

- 3. Include other header files that define macros and variables that the function uses.
- 4. Define the function parameter list in the order that matches the parameter passing convention specified in the METHOD specification of the CREATE TYPE statement.

For more information, see [UDM Parameter List](#).

- 5. Implement the function and set the result to the appropriate value.

If the function is a constructor UDM for a structured UDT or an instance UDM that provides cast, ordering, or transform functionality, the implementation must satisfy certain requirements.

FOR details on implementing a C/C++ function that ...	SEE ...
is a constructor UDM	<a href="#">Constructor UDMs</a> .
provides cast functionality for a UDT	<a href="#">UDMs That Provide Cast Functionality for a UDT</a> .
provides ordering functionality for a UDT	<a href="#">UDMs That Provide Ordering Functionality for a UDT</a> .
provides transform functionality for a UDT	<a href="#">UDMs That Provide Transform Functionality for a UDT</a> .

- 6. If the function detects an error, set the:
  - `sqlstate` argument to an SQLSTATE exception or warning condition before the function exits.  
For more information, see [Returning SQLSTATE Values](#).
  - `error_message` string to the error message text. The characters must be inside the LATIN character range. The string is initialized to a null-terminated string on input.
- 7. If the method uses parameter style SQL, set the `indicator_result` argument.

IF the result is ...	THEN set the <code>indicator_result</code> argument to ...
NULL	-1.
not NULL	0.

**Example: Basic UDM**

Here is an example of a simple C function that implements an instance UDM for a distinct UDT. The function uses the value of the UDT, which is in meters, and returns the equivalent value in inches. The UDM uses parameter style TD\_GENERAL.

```

/***** C source file name: to_inches.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
```

```

#include <string.h>

void meter_toInches( UDT_HANDLE  *meterUdt,
                    FLOAT        *result,
                    char          sqlstate[6])
{
    FLOAT value;
    int length;

    /* Get meter's value. */
    FNC_GetDistinctValue(*meterUdt, &value, sizeof_FLOAT, &length);

    /* Convert meters to inches and set the result value. */
    *result = value * 3.28 * 12;
}

```

### Using UDT Handles in a UDM

Teradata provides library functions that a UDM can use with UDT\_HANDLE input arguments and return values.

IF you want to ...	THEN use this library function ...
get the value of a distinct type	FNC_GetDistinctValue
set the value of a distinct type	FNC_SetDistinctValue
get the locator for a distinct type that represents a LOB	FNC_GetDistinctInputLob
	FNC_GetDistinctResultLob
get the attribute value of a structured type	FNC_GetStructuredAttribute
set the attribute value of a structured type	FNC_SetStructuredAttribute
get the locator for a structured type LOB attribute	FNC_GetStructuredInputLobAttribute
	FNC_GetStructuredResultLobAttribute

Details on how to use the library functions appear in other sections of this document.

FOR details on how to ...	SEE ...
access the value of a distinct UDT	<a href="#">Accessing the Value of a Distinct UDT.</a>
access the attribute values of a structured UDT	<a href="#">Accessing the Attribute Values of a Structured UDT.</a>
set the value of a distinct UDT result	<a href="#">Setting the Value of a Distinct UDT Result.</a>
set the attribute values of a structured UDT result	<a href="#">Setting the Attribute Values of a Structured UDT Result.</a>

## UDMs That Provide Cast Functionality for a UDT

If the function is an instance UDM that provides cast functionality for a UDT, then the C/C++ function must satisfy the following requirements.

IF the method implements cast functionality that casts ...		THEN the C/C++ function must set the ...
FROM ...	TO ...	
another UDT or predefined type	a structured UDT	attribute values of the UDT result using the value of the input argument.
	a distinct UDT	value of the UDT result using the value of the input argument.
a structured UDT	another UDT or predefined type	value of the result using the attribute values of the UDT input argument.
a distinct UDT		value of the result using the value of the UDT input argument.

Vantage automatically generates cast functionality between a distinct type and its predefined source type. You can create additional methods for casting between a distinct type and other predefined data types or UDTs.

For examples of how to implement UDMs that provide cast functionality, see [UDM Code Examples](#).

## UDMs That Provide Ordering Functionality for a UDT

If the function is an instance UDM that provides ordering functionality for comparing two UDTs, the implementation must satisfy the following requirements.

IF the UDT is ...	THEN the C/C++ function must use the ...
structured	attribute values of the UDT to set the result to a value that Vantage uses for comparisons.
distinct	value of the UDT to set the result to a value that Vantage uses for comparisons.

Vantage automatically generates ordering functionality when you create a distinct type where the source data type is not a LOB. However, you can drop the functionality and provide your own.

For an example of how to implement a UDM that provides ordering functionality, see [UDM Code Examples](#).

## UDMs That Provide Transform Functionality for a UDT

If the function is an instance UDM that provides transform functionality for exporting a UDT from the server, the implementation must satisfy the following requirements.

IF the UDT is ...	THEN the C/C++ function must transform the ...
structured	attribute values of the UDT into an appropriate value for the predefined type result.

IF the UDT is ...	THEN the C/C++ function must transform the ...
distinct	value of the UDT into an appropriate value for the predefined type result.

Although Vantage automatically generates transform functionality when you create a distinct type, you can drop the functionality and provide your own.

For an example of how to implement a UDM that provides transform functionality, see [UDM Code Examples](#).

## Constructor UDMs

If the function is a constructor UDM for a structured UDT, the C/C++ function must:

- set the attribute values of the result UDT to the attribute values of UDT on which the UDM is invoked
- set the attribute values of the result UDT to the values of corresponding input arguments

For an example of how to implement a constructor method, see [UDM Code Examples](#).

## UDMs That Use CLOB or BLOB Arguments

UDMs can define CLOB or BLOB input and output arguments, passing them by locator.

The guidelines for implementing UDMs that use CLOB or BLOB arguments are similar to the guidelines for implementing UDFs that use CLOB or BLOB arguments. For details, see [Defining Functions that Use LOB Types](#).

## UDMs That Use Period Arguments

UDMs can define Period input and output arguments, passing them by PDT handle.

The guidelines for defining UDMs that use Period arguments are similar to the guidelines for defining scalar UDFs that use Period arguments. For details, see [Defining Functions that Use Period Types](#).

## UDMs That Use ARRAY Arguments

UDMs can define ARRAY input and output arguments, passing them by ARRAY handle.

The guidelines for defining UDMs that use ARRAY arguments are similar to the guidelines for defining scalar UDFs that use ARRAY arguments. For details, see [Defining Functions that Use ARRAY Types](#).

## UDMs That Use TD\_ANYTYPE Arguments

You can define UDMs with input and output parameters that are of TD\_ANYTYPE data type.

The guidelines for defining UDMs that use TD\_ANYTYPE arguments are similar to the guidelines for defining scalar UDFs that use TD\_ANYTYPE arguments. For details, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).

## Returning SQLSTATE Values

The parameter list of a UDM includes an output character string for returning the SQLSTATE result code value.

## C Data Type

The following table defines the C data type that corresponds to the SQLSTATE result code variable.

Result Code Variable	C Data Type
SQLSTATE	char sqlstate[6]

## SQLSTATE Values

The first five characters of the *sqlstate* output character string have a format of 'ccsss', where *cc* is the class and *sss* is the subclass. The last character of the string is a binary 0, or C string terminator.

For more information on the valid settings that a UDM can return for the SQLSTATE result code, see [SQLSTATE Values](#).

## Initial Value

The *sqlstate* output character string is initialized to '00000' (five zero characters), which corresponds to a success condition. Therefore, you do not have to set the value of the *sqlstate* output argument for a normal return.

## Display Format

If a UDM returns an SQLSTATE value other than success, a BTEQ session displays an error.

IF the SQLSTATE category is ...	THEN the display format is ...
not a warning	<pre>*** Failure 7504 in UDF/XSP  databasename.udmname: SQLSTATE  ccsss: &lt;text&gt;</pre> <p>where:</p> <ul style="list-style-type: none"> <li>7504 is the Teradata designated error code for user-defined functions, user-defined methods, and external stored procedures</li> <li><i>udmname</i> is the name of the UDM</li> <li><i>ccsss</i> is the value that the UDM sets the SQLSTATE output argument to, according to the table in <a href="#">SQLSTATE Values</a>.</li> <li><i>&lt;text&gt;</i> is the value of the error message output argument, if the UDM uses parameter style SQL</li> </ul>
warning	<pre>*** Warning: 7505 in UDF/XSP databasename.udmname: SQLSTATE 01H xx: &lt;text&gt;</pre> <p>where:</p> <ul style="list-style-type: none"> <li>7505 is the Teradata designated warning code for user-defined functions, user-defined methods, and external stored procedures</li> <li><i>udmname</i> is the name of the UDM</li> </ul>

IF the SQLSTATE category is ...	THEN the display format is ...
	<ul style="list-style-type: none"><li>• 01H xx is the value that the UDM sets the SQLSTATE output argument to, according to values for the warning category in the table in <a href="#">SQLSTATE Values</a>.</li><li>• &lt;text&gt; is the value of the error message output argument, if the UDM uses parameter style SQL</li></ul>

**Example: Returning the SQLSTATE Result Code Value**

Consider the following C function that implements an instance method for a structured UDT that has an attribute called *social\_security\_number*:

```
void encrypt( UDT_HANDLE      *personalUdt,
              VARCHAR_LATIN  *result,
              char             sqlstate[6])
{
    ...
}
```

You can use the *sqlstate* argument to return the SQLSTATE result code value. For example, if the *social\_security\_number* attribute value of the UDT argument is not a valid value, you can set the value of the *sqlstate* argument to return a data exception:

```
strcpy(sqlstate, "U0005");
```

In a BTEQ session, the exception condition appears in the following format:

```
*** Failure 7504 in UDF/XSP employee.encrypt: SQLSTATE U0005:
```

**Related Information**

FOR more information on ...	SEE ...
using a warning condition to debug a method	<a href="#">Debugging a UDM.</a>

**Installing the UDM**

After you write and test a UDM, you can install it on the server.



**Note:**

In general, you should not create UDFs, UDMs, or external stored procedures in Teradata system databases such as SYSLIB or SYSUDTLIB. These databases are primarily used for Teradata system UDFs, UDTs, UDMs, and external stored procedures only, and they usually contain a large number of these system external routines. Every time you create, alter, or drop your external routine in these databases, Teradata must relink your routine to all the objects of the system external routines. In addition, to execute your routine, Teradata must load all the shared libraries referenced by the system external routines, and these libraries may not be related to your routine. This is very inefficient. However, note that there are cases where you have to create your UDF in a system database. For example, UDFs used for row level security must reside in the SYSLIB database.

**CREATE METHOD Statement**

To identify the file name and location of the source code and install it on the server, use the CREATE METHOD statement.

The method is compiled, linked to the dynamic linked library (DLL or SO) associated with the database in which the method resides, and distributed to all database nodes in the system.

Note that all UDMs that are defined in a specific database are linked into a single dynamically linked library.

**Default and Temporary Paths**

The default and temporary paths that Vantage uses to manage UDMs during creation and execution are the same default and temporary paths that are used for UDFs and external stored procedures. For example, Vantage uses the same temporary directory to compile UDFs, UDMs, and external stored procedures.

For more information on default and temporary paths, see [Default and Temporary Paths](#).

**Specifying Source File Locations**

The EXTERNAL clause of the CREATE METHOD statement specifies the name and path of the source code file.

CREATE METHOD Clause	Description
EXTERNAL	<p>Use the EXTERNAL clause when the method source is in the current or default directory on the client, and no other files need to be included.</p> <p>The source name is the name that immediately follows the CREATE METHOD keywords. If the client is...</p> <ul style="list-style-type: none"> <li>workstation-attached, then BTEQ adds appropriate file extensions to the source name to locate the source file.</li> <li>mainframe-attached, then the source name must be a DDNAME file name.</li> </ul> <p>Here is an example for a source code file named toInches.c:</p> <pre>CREATE METHOD toInches() RETURNS FLOAT</pre>

CREATE METHOD Clause	Description
	<p>FOR meter EXTERNAL;</p>
EXTERNAL NAME <i>method_name</i>	<p>Use the EXTERNAL <i>method_name</i> clause when the source is in the current or default directory on the client, and no other files need to be included. The source name is the same as <i>method_name</i>. If the client is...</p> <ul style="list-style-type: none"> <li>• workstation-attached, then BTEQ adds appropriate file extensions to <i>method_name</i> to locate the source file.</li> <li>• mainframe-attached, then <i>method_name</i> must be a DDNAME file name.</li> </ul> <p>Here is an example for a source code file named <i>meter_toInches.c</i>:</p> <pre>CREATE METHOD toInches() RETURNS FLOAT FOR meter EXTERNAL NAME meter_toInches;</pre>
EXTERNAL NAME ' <i>string</i> '	<p>Use '<i>string</i>' to specify names and locations of:</p> <ul style="list-style-type: none"> <li>• Function source, include header files, object files, libraries, and packages on the server.</li> <li>• Function source, include header files, and object files on the client.</li> </ul> <p>If the client is...</p> <ul style="list-style-type: none"> <li>• workstation-attached, then if necessary, BTEQ adds appropriate file extensions to the names to locate the files.</li> <li>• mainframe-attached, then the names must be DDNAME file names.</li> </ul> <p>Here is an example:</p> <pre>CREATE METHOD toInches() RETURNS FLOAT FOR meter EXTERNAL NAME 'CS!toInches!udmsrc/toInches.c!F!meter_toInches';</pre> <p>where:</p> <ul style="list-style-type: none"> <li>• ! in '<i>string</i>' specifies a delimiter.</li> <li>• C in '<i>string</i>' specifies that the source is obtained from the client.</li> <li>• S in '<i>string</i>' specifies that the information between the following two sets of delimiters identifies the name and location of a C or C++ function source file.</li> <li>• toInches in '<i>string</i>' specifies the name, without the file extension, that the server uses to compile the source.</li> <li>• udmsrc/toInches.c in '<i>string</i>' specifies a relative path (udmsrc) for the source file (toInches.c).</li> <li>• F in '<i>string</i>' specifies that the information after the next delimiter identifies the C or C++ function name.</li> <li>• meter_toInches in '<i>string</i>' specifies the C or C++ function name.</li> </ul>

For more information on CREATE METHOD and the EXTERNAL clause, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Source File Locations for ODBC

If you use ODBC, you can only create UDMs from files that are stored on the server.

## Source File Locations for JDBC

Using Teradata Driver for the JDBC Interface, you can create UDMs from files that are located on the Teradata server, or from resources located on the client.

A client-side UDM source file must be available as a resource in the class path. The Teradata JDBC driver can load the resource from the class path and transfer it to the server node without directly accessing the client file system.

## Specifying the C/C++ Function Name

In addition to specifying the location of the source code, the `EXTERNAL` clause in the `CREATE METHOD` statement identifies the C/C++ function name that appears in the C/C++ function declaration or the function entry name when the C/C++ object is provided instead of the C/C++ source.

IF CREATE METHOD specifies this clause ...	THEN ...
EXTERNAL	<p>The C/C++ function name must match the name that follows the <code>CREATE METHOD</code> keywords.</p> <p>Consider the following <code>CREATE METHOD</code> statement:</p> <pre>CREATE METHOD toInches() RETURNS FLOAT FOR meter EXTERNAL;</pre> <p>The C or C++ function name must be <code>toInches</code>.</p> <p>If the client is mainframe-attached, then the C/C++ function name must be the DDNAME for the source.</p>
EXTERNAL NAME <i>method_name</i>	<p>the C/C++ function name must match <i>method_name</i>.</p> <p>Consider the following <code>CREATE METHOD</code> statement:</p> <pre>CREATE METHOD toInches() RETURNS FLOAT FOR meter EXTERNAL NAME meter_toInches;</pre> <p>The C/C++ function name must be <code>meter_toInches</code>.</p> <p>If the client is mainframe-attached, then <i>method_name</i> must be the DDNAME for the source.</p>
EXTERNAL NAME 'string'	<p>'string' can include the F option to specify the C/C++ function name.</p> <p>Consider the following <code>CREATE METHOD</code> statement:</p>

IF CREATE METHOD specifies this clause ...	THEN ...
	<pre>CREATE METHOD toInches() RETURNS FLOAT FOR meter EXTERNAL NAME 'CS!toInches!udmsrc/toInches.c!F!meter_toInches';</pre> <p>The C/C++ function name must be meter_toInches. If '<i>string</i>' does not include the F option, then the C/C++ function name must match the name that follows the CREATE METHOD keywords.</p>

## Specifying Nonstandard Include Files

If a UDM includes a nonstandard header file, the EXTERNAL clause in the CREATE METHOD statement must specify the name and path of the header file.

Consider the following UDM that includes the header file `ssn.h`:

```

/***** C source file name: p_ssn.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include "ssn.h"

void encrypt( UDT_HANDLE    *personalUdt,
              VARCHAR_LATIN *result,
              char           sqlstate[6])
{
    ...
}
```

Here is an example of CREATE METHOD that specifies the name and path of the nonstandard header file:

```

CREATE METHOD encrypt()
RETURNS VARCHAR(64)
FOR personal
EXTERNAL NAME
'CI!ssn!udm_home/ssn.h!CS!encrypt!udm_home/p_ssn.c!F!encrypt';
```

where:

This part of the string that follows <b>EXTERNAL NAME ...</b>	Specifies ...
!	a delimiter.
C	that the header file is obtained from the client.
I	that the information between the following two sets of delimiters identifies the name and location of an include file (.h).
ssn	the name, without the file extension, of the header file.
udm_home/ssn.h	the path and name of the header file on the client.
C	that the source is obtained from the client.
S	that the information between the following two sets of delimiters identifies the name and location of a C or C++ function source file.
encrypt	the name, without the file extension, that the server uses to compile the source.
udm_home/p_ssn.c	the path and name of the source file.
F	that the information after the next delimiter identifies the C or C++ function name.
encrypt	the C or C++ function name.

For more information on installing libraries, see the information about CREATE METHOD in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Related Information

FOR more information on ...	SEE ...
the CREATE METHOD statement	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the privileges that apply to UDTs and UDMs	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.
Linux limitations that affect name and number of external routines in a database	<a href="#">Argument list too long.</a>

## Debugging a UDM

When debugging a UDM, be sure your tests include the following:

- Limit checks on input values
- Limit checks on return values
- Proper handling of NULLs

- Division by zero
- All memory acquired by *malloc* is freed
- All open operating system handles are released and closed

The Teradata C/C++ UDF Debugger allows user-defined methods to be debugged within the database on a development or test system. For more information on the Teradata C/C++ UDF Debugger, see [C/C++ Command-line Debugging for UDFs](#).

### Forcing an SQL Warning Condition

You can force an SQL warning condition at the point in the code where the method appears to have problems. To force the warning, set the *sql/state* return argument to '01H xx', where you choose the numeric value of xx.

If you use parameter style SQL, you can also set the *error\_message* return argument to return up to 256 SQL\_TEXT characters.

The warning is issued as a return state of the UDM. The warning does not terminate the transaction; therefore, you must set return arguments to valid values that the transaction can use.

Only one warning can be returned per invocation.

You force an SQL warning condition in a UDM in the same manner you force an SQL warning condition in a UDF. For an example of how to set the SQLSTATE result code and error message return argument in a UDF, see [Forcing an SQL Warning Condition](#).

### Using Trace Tables

If debugging outside the database is not sufficient, you can use trace tables to get trace diagnostic output.

UDMs use trace tables in the same manner as UDFs use trace tables. For a description of how to debug UDFs using trace tables, see [Debugging Using Trace Tables](#).

## Resolving UDF Server Setup Errors

When an external routine like an UDF, UDM, or external stored procedure is called, a UDF server process is acquired from the UDF server pool to execute the external routine. If there are no UDF server processes in the pool or if all of the processes in the pool are busy, then the system tries to start a new UDF server process for the request.

The startup of the new UDF server usually takes some time, especially if the UDF server is for executing Java external routines, or if the system is very busy. If the new UDF server cannot be started within the default time limit, the query that contains the UDF, UDM, or external procedure call is aborted, and you may receive a 7583 error indicating that the UDF server setup encountered a problem. The system log may also record a 7820 error specifying that the UDF server could not stay up long enough for initialization.

If you are experiencing these errors, you can contact Teradata Support Center personnel to adjust the time limit allowed for starting a new UDF server process. For details, see the information about the Cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## UDM Invocation

You invoke a constructor method to create and initialize a new instance of a structured UDT and you invoke an instance method on an existing instance of a distinct or structured UDT.

### Argument List

The arguments in the method invocation must appear as comma-separated expressions in the same order as the parameters in the METHOD specification of the CREATE TYPE statement.

The arguments in the method invocation must be compatible with the parameter declarations in the method definition of an existing method, and must fit into the compatible type without a possible loss of information. For example, a BYTEINT argument in a method invocation is compatible with an INTEGER parameter declaration in the method definition, and also fits into the INTEGER type without any loss of information.

To pass an argument that is not compatible with the corresponding parameter type, explicitly convert the argument to the proper type in the method invocation.

The NULL keyword is compatible with a parameter of any data type. For more information on the behavior of NULL as a literal argument, see [Argument Behavior](#).

For information on compatible types and the precedence rules, see [UDM Parameters](#).

### The RETURNS and RETURNS STYLE Clauses

When invoking a UDM that is defined with a TD\_ANYTYPE return parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clauses to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

The RETURNS or RETURNS STYLE clause is not mandatory as long as the method also includes a TD\_ANYTYPE input parameter. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD\_ANYTYPE input argument is used to determine the return type of the TD\_ANYTYPE return parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.

Note that you must enclose the UDM invocation in parenthesis if you use the RETURNS or RETURNS STYLE clauses.

### Method Selection

When you use a METHOD specification in a CREATE TYPE statement, the database takes the UDT name, method name, and method parameter list to form an internal method name and parameter list that looks like this:

```
method_name(UDT_name, parameter_1 ... , parameter_n)
```

Similarly, when you invoke a method on a UDT, the database uses the UDT name, method name, and method argument list to form an internal method name and argument list that looks like this:

```
method_name(UDT_name, argument_1 ... , argument_n)
```

Vantage searches the database for an internal method and parameter list that matches the internal method and argument list in the method invocation.

IF such a method ...	THEN ...
exists, and the arguments in the method invocation are compatible with the method parameters and follow the order of precedence	the search stops. If several methods have the same name, Vantage follows the same rules to determine which method to invoke that it uses to determine which function to invoke if several UDFs have the same name. For details, see <a href="#">Overloaded Function Invocation</a> .
does not exist, or the arguments in the method invocation are not compatible with the method parameters	the statement returns an error.

For the rules of compatibility precedence, see [UDM Parameters](#).

## Overloaded Method Invocation

If several UDMs have the same name, Vantage follows the same rules to determine which method to invoke that it uses to determine which function to invoke if several UDFs have the same name.

For details, see [Overloaded Function Invocation](#).

## Related Information

FOR more information on ...	SEE ...
example SQL statements that invoke instance and constructor methods	<ul style="list-style-type: none"> <li><a href="#">UDM Code Examples</a>.</li> <li><i>Teradata Vantage™ - SQL Operators and User-Defined Functions</i>, B035-1210.</li> </ul>
instance and constructor method invocation syntax	<i>Teradata Vantage™ - SQL Operators and User-Defined Functions</i> , B035-1210.
using the NEW expression to construct UDT instances	
restrictions that apply to instance and constructor method invocation	

# Protected Mode Execution

## Protected Mode Execution Option

The ALTER METHOD statement provides an execution option that controls whether Vantage invokes the method directly or runs the method indirectly as a separate process.



The option applies to methods that were created without specifying the EXTERNAL SECURITY clause in the CREATE METHOD or REPLACE METHOD statement. Methods that specify the EXTERNAL SECURITY clause are executed using separate secure server processes.

IF ALTER METHOD specifies ...	THEN Vantage ...
EXECUTE PROTECTED	runs the method indirectly as a separate process. If the method fails during execution, the transaction fails.
EXECUTE NOT PROTECTED	invokes the method directly.

## NOTICE

If the ALTER METHOD statement specifies EXECUTE NOT PROTECTED, and the UDM fails during execution, the database software will probably restart.

Only an administrator, or someone with sufficient privileges, can use the ALTER METHOD statement.

## Choosing the Correct Execution Option

Use the following table to choose the correct execution option for UDMs.

IF ...	THEN use ...
you are in the development phase and are debugging a UDM	EXECUTE PROTECTED.
the UDM opens a file or uses another operating system resource that requires tracking by the operating system	EXECUTE PROTECTED. Running such a UDM in nonprotected mode could interfere with the proper Teradata system operation.
the UDM does not use any operating system resources	EXECUTE NOT PROTECTED. Running a UDM in nonprotected mode speeds up processing considerably. Use this option only after thoroughly debugging the UDM and making sure it produces the correct output.

## Related Information

FOR more information on ...	SEE ...
protected mode process and server administration	<a href="#">Protected Mode Process and Server Administration for C/C++ External Routines.</a>
ALTER METHOD and the EXECUTE PROTECTED option	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the privileges associated with UDTs	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Argument Behavior

### Truncation of Character String Arguments

The session transaction mode affects character string truncation.

IF the session transaction mode is ...	THEN an input character string that requires truncation is truncated ...
Teradata	without reporting an error. Truncation on Kanji1 character strings containing multibyte characters might result in truncation of one byte of the multibyte character.
ANSI	of excess pad characters without reporting an error. Truncation of other characters results in a truncation exception.

The normal truncation rules apply to a result string. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

### Behavior When Using NULL as a Literal Argument

The RETURNS NULL ON NULL INPUT and CALLED ON NULL INPUT options in the METHOD specification of the CREATE TYPE statement determine what happens if the NULL keyword is used as any of the input arguments.

IF an input argument is NULL and the corresponding CREATE TYPE statement specifies ...	AND the parameter style is ...	THEN ...
RETURNS NULL ON NULL INPUT	SQL or TD_GENERAL	the method is not evaluated and the result is always NULL.
CALLED ON NULL INPUT	SQL	the method is invoked with the appropriate indicators set to the null indication.
	TD_GENERAL	an error is reported.

If the METHOD specification in the CREATE TYPE statement does not specify either RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT, the default is CALLED ON NULL INPUT.

NULL as a literal argument is compatible with a parameter of any data type. For example, consider the following *address* UDT and *map\_it* method:

```
CREATE TYPE address
  AS ( street VARCHAR(20),
        zip CHAR(5) )
  NOT FINAL
  INSTANCE METHOD map_it( CHAR(4), INTEGER )
  RETURNS INTEGER
```

```

    SPECIFIC mapit
    LANGUAGE C
    PARAMETER STYLE SQL
    DETERMINISTIC
    NO SQL;

CREATE METHOD map_it( map_type CHAR(4), code INTEGER )
RETURNS INTEGER
FOR address
EXTERNAL NAME 'CS!map_it!udm_src/map_it.c!F!mapit';

```

You can successfully use the NULL keyword as any method argument:

```

CREATE TABLE locations( id INTEGER, details address );

SELECT details.map_it('4415', NULL) FROM locations;
SELECT details.map_it(NULL, NULL) FROM locations;

```

Passing the NULL keyword as an argument to overloaded methods can result in errors unless Vantage can identify which method to invoke without ambiguity. For details, see [Calling a Function That is Overloaded](#).

## Overflow and Numeric Arguments

To avoid numeric overflow conditions, the C/C++ function should define a decimal data type as big as it can handle.

If the assignment of the value of an input or output numeric argument would result in a loss of significant digits, a numeric overflow error is reported.

For example, consider a instance method that takes a DECIMAL(2,0) argument:

```

CREATE TYPE circle
  AS ( x INTEGER,
        y INTEGER,
        radius INTEGER )
  NOT FINAL
  INSTANCE METHOD smldec( DECIMAL(2,0) )
  RETURNS INTEGER
  SPECIFIC smldec
  LANGUAGE C
  PARAMETER STYLE SQL
  DETERMINISTIC
  NO SQL;

CREATE METHOD smldec( DECIMAL(2,0) )

```

```
RETURNS INTEGER  
FOR circle  
EXTERNAL NAME 'CS!smldec!udm_src/smldec.c!F!smldec';
```

Passing a number with a maximum of two digits is successful:

```
SELECT circle_column.smldec(99) FROM circle_tbl WHERE c_id = 100;
```

An attempt to pass a number larger than 99 or smaller than -99 would result in a loss of significant digits.

```
SELECT circle_column.smldec(100) FROM circle_tbl WHERE c_id = 100;
```

```
Failure 2616 Numeric overflow occurred during computation.
```

Any fractional numeric data that is passed or returned that does not fit as it is being assigned is rounded according to Teradata rounding rules. For more information on rounding, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

# Java User-Defined Functions

SQL provides a set of useful functions, but they might not satisfy all of the particular requirements you have to process your data.

User-defined functions (UDFs) allow you to extend SQL by writing your own functions in the Java programming language, installing them on the database, and then using them like standard SQL functions.

For details on writing your own functions in the C or C++ programming language, see [C/C++ User-Defined Functions](#)

## UDF Types

Teradata supports three types of Java UDFs.

Java UDF Type	Description
Scalar	Scalar functions take input arguments and return a single value result. Some examples of standard SQL scalar functions are CHARACTER_LENGTH, POSITION, and SUBSTRING. You can use a scalar function in place of a column name in an expression. When Vantage evaluates the expression, it invokes the scalar function. No context is retained after the function completes.
Aggregate	Aggregate functions produce summary results. They differ from scalar functions in that they take grouped sets of relational data, make a pass over each group, and return one result for the group. Some examples of standard SQL aggregate functions are AVG, SUM, MAX, and MIN.  Vantage invokes an aggregate function once for each item in the group, passing the detail values of a group through the input arguments. To accumulate summary information, an aggregate function must retain context each time it is called.  You do not need to understand or worry about how to create a group, or how to code an aggregate UDF to deal with groups. Vantage automatically takes care of all of those difficult aspects. You only need to write the basic algorithm of combining the data passed in to produce the desired result.
Table	A table function is invoked in the FROM clause of an SQL SELECT statement and returns a table a row at a time in a loop to the SELECT statement. The function can produce the rows of a table from the input arguments passed to it or by reading an external file or message queue. The number of columns in the rows that a table function returns can be specified dynamically at runtime in the SELECT statement that invokes the table function.

## Java Development Environment

The topics that this section discusses do not assume that you have any specific development environment beyond a Java SDK or JRE.

If you use Eclipse as a Java IDE, you can develop scalar Java UDFs using the Teradata Plug-In for Eclipse, available from [Teradata Downloads](#). For details about creating Java UDFs and managing JAR files using the Teradata Plug-In for Eclipse, see *Teradata Plug-In for Eclipse Release Definition*, also available from Teradata Downloads.

## Overall Procedure Synopsis

### System Requirements

Before you can run Java UDFs, your system must meet certain requirements. For details, see [System Requirements](#).

### Procedure

Here is a synopsis of the steps you take to develop, compile, install, and invoke a Java UDF. You can find details for each step in subsequent sections.

1. Write, test, and debug the Java source code for the UDF outside of the database and place the resulting class or classes in a JAR or ZIP file (collectively called *archive* files).
2. Call the SQLJ.INSTALL\_JAR external stored procedure to register the archive file and its classes with the database, providing an SQL identifier for the JAR or ZIP file.

---

**Note:**

Files with .zip extensions are treated as .jar files, and are registered in the database as JAR objects.

---

3. Determine the level of access to operating system services that the UDF requires.

IF the UDF ...	THEN ...
does not access local files and does not perform any restricted actions that require special permissions	the UDF can run in protected execution mode as a thread of the Java hybrid server. Vantage creates one Java hybrid server for each node, and the server provides multiple threaded execution of Java external routines to all AMPs and PEs on the node.
accesses local files or performs actions that ordinary operating system users have permissions for	For security, the hybrid server and all of its resources, such as shared map files and events, use 'tdatuser' as the owner, a local operating system user that the database installation process creates.
requires access to specific resources that require special permissions	use CREATE AUTHORIZATION or REPLACE AUTHORIZATION to create a context that identifies a native operating system user and allows the UDF to access resources by using its own secure server under the authorization of that user.

4. Use CREATE FUNCTION or REPLACE FUNCTION with options that provide specific information about the Java UDF.

Option	Description
LANGUAGE JAVA	Identifies the source code language of the UDF.
PARAMETER STYLE JAVA	Indicates the parameter style.
EXTERNAL NAME	Provides the registered name of the JAR or ZIP file, Java class within the JAR or ZIP file, and Java method within the class to execute when the UDF is invoked in an SQL statement.
EXTERNAL SECURITY (Optional)	Associates execution of the UDF with the context created by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement in the previous step.

5. Test the UDF until you are satisfied it works correctly.
6. Use GRANT to grant privileges to users who are authorized to use the UDF.

## Related Information

FOR more information on ...	SEE ...
creating a UDF	related topics in this document.
debugging a UDF	<a href="#">Debugging a User-Defined Function.</a>
<ul style="list-style-type: none"> <li>protected mode function execution</li> <li>tdatuser operating system user</li> </ul>	<a href="#">Administration.</a>
code examples for scalar, aggregate, and table UDFs	<a href="#">UDF Code Examples.</a>
<ul style="list-style-type: none"> <li>CREATE FUNCTION</li> <li>REPLACE FUNCTION</li> </ul>	<ul style="list-style-type: none"> <li><a href="#">Defining the SQL Function.</a></li> <li><i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li><i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
<ul style="list-style-type: none"> <li>CREATE AUTHORIZATION</li> <li>REPLACE AUTHORIZATION</li> </ul>	<ul style="list-style-type: none"> <li><i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li><i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>

## UDF Source Code Development

You develop the body of a UDF using the Java programming language.

You can use any class name or method name to implement the Java UDF. Later, when you use the CREATE FUNCTION or REPLACE FUNCTION statement to create an SQL identifier for the UDF, you specify the class and method names so that the database knows what to execute when an SQL statement specifies the UDF.

---

**Note:**

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

---

## Source Code Contents

The topics that follow provide details about the contents of the source code for a UDF. In general, you must provide a Java class that defines a method that Vantage executes when the UDF is called. The parameters that the method takes match the parameters specified in the CREATE FUNCTION or REPLACE FUNCTION statement.

## Class Fields

Class, or static, fields in Java classes may only represent constants and must be declared with the *final* modifier.

## Resource Access

Java UDFs can access resources, such as local files, if required. The CREATE FUNCTION or REPLACE FUNCTION statement for the UDF determines which resources the UDF has access to.

IF the CREATE/ REPLACE FUNCTION statement ...	THEN the UDF ...
specifies the EXTERNAL SECURITY clause	can access specific resources that require special permissions, in addition to accessing local files or performing actions that ordinary operating system users have permissions for.  Teradata uses a separate secure server to execute the UDF under the authorization of a specific native operating system user established by a CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement.
does not specify the EXTERNAL SECURITY clause	runs in protected execution mode and can access local files or perform actions that ordinary operating system users have permissions for.  Vantage executes the UDF as a thread of a hybrid server that runs under the authorization of the 'tdatuser' operating system user.

For more information on Java server administration, see [Server Administration for Java External Routines](#).



## Teradata Application Classes

Teradata provides Java application classes that you can use for UDF development. The following table shows some of the classes that are available.

Class	Description
com.teradata.fnc.Blob	For UDFs that use an SQL BLOB type parameter or return type
com.teradata.fnc.Clob	For UDFs that use an SQL CLOB type parameter or return type
com.teradata.fnc.DbsInfo	For UDFs that need to obtain session information related to the current execution of the UDF
com.teradata.fnc.TraceObj	Provides methods for UDFs that write trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement for debugging purposes
com.teradata.fnc.QueryBand	Provides methods for UDFs that need to access query band information for a session, transaction, and/or profile
com.teradata.fnc.Phase	Classes used by an aggregate function
com.teradata.fnc.Context	
com.teradata.fnc.Tbl	Classes used by a table function
com.teradata.fnc.AmplInfo	
com.teradata.fnc.NodeInfo	

For a list of all available classes and their descriptions, see [Java Application Classes](#)

## Attempting to Exit the Java Virtual Machine

A Java UDF must not call `System.exit()` or any similar methods. Any attempt to terminate or halt the currently running Java Virtual Machine by calling `System.exit()` or similar methods generates an error.

## Compiling the Source Code

To compile the source code for your Java routine, you need `javFnc.jar`, the Teradata Java external stored procedure and UDF runtime library.

IF you compile the source code on ...	THEN ...
your client system	download the runtime library from <a href="#">Teradata Downloads</a> .
the database system	the runtime library is already available.

Add the path to the runtime library to your class path (or use the `-classpath` option of the Java compiler). For details on the location of the runtime library on a database system, see [Class Path](#).

For example, suppose you downloaded javFnc.jar to C:\java\_udf on your Windows client. You can compile the source code in UDFExample.java as follows:

```
javac -classpath C:\java_udf\javFnc.jar UDFExample.java
```

The version of the class file that you generate when you compile the Java source code must be compatible with the JRE on the database system.

For example, suppose the database uses JRE 1.8 and your client system uses JDK 8.0. To compile the source code in UDFExample.java on your client system, you can use the -target option of the Java compiler.

```
javac -target 1.8 -classpath C:\java_udf\javFnc.jar UDFExample.java
```

To determine the version of the JRE on the database system, use the cufconfig utility to see where the JRE was installed. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Class and Method Names

The name of the class and method in the source code follows the Java naming conventions.

When you use the CREATE FUNCTION or REPLACE FUNCTION statement to create an SQL identifier for the UDF, you specify the name of the class and method in the EXTERNAL NAME clause. For more information, see [Defining the SQL Function](#).

## Parameter List and Return Value

The list of parameters for a method that implements a UDF is very specific. It includes the input parameters that Vantage passes when the UDF appears in an SQL statement.

A UDF can have 0 to 128 input parameters.

---

### Note:

A Java UDF cannot have an ARRAY type parameter where the base type is a nested structured UDT.

---

## Default Mapping Convention of Parameter Types

The data types that you use in the parameter list of the Java method map to the SQL data types in the parameter list of the CREATE FUNCTION or REPLACE FUNCTION statement.

Consider the following CREATE FUNCTION statement for a scalar UDF:

```
CREATE FUNCTION factorial
  (x INTEGER)
  RETURNS INTEGER
```

```
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'JarUDF:UDFExample.fact';
```

The parameter list specifies that the SQL data type of *x* is INTEGER. The signature of the *fact* method that implements the UDF looks like this:

```
public static int fact( int x ) { ... }
```

The default mapping convention is simple mapping, where SQL data types map to Java primitives. If no Java primitive can adequately map to an SQL type, then the default mapping convention is object mapping, where SQL data types map to Java classes.

Consider a factorial UDF that takes a DECIMAL value:

```
CREATE FUNCTION factorial
  (x DECIMAL(8,2))
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'JarUDF:UDFExample.fact';
```

Because the DECIMAL type does not map adequately to a Java primitive, the DECIMAL type maps to `java.math.BigDecimal`. The signature of the *fact* method that implements the UDF looks like this:

```
public static int fact( BigDecimal x ) { ... }
```

For details on how SQL data types map to Java data types, see [SQL Data Type Mapping](#).

## Overriding the Default Mapping of Parameters

For UDFs that allow NULL as input arguments (where the CREATE FUNCTION or REPLACE FUNCTION statement specifies the CALLED ON NULL INPUT clause), simple mapping to Java primitives is not appropriate because they cannot represent NULLs.

To override the default mapping, the EXTERNAL NAME clause in the CREATE FUNCTION or REPLACE FUNCTION statement must explicitly specify the mapping in the parameter list of the Java method. For an example that shows how to override the default mapping, see [Example: Overriding Default Parameter Mapping to Handle NULLs](#).

## Default Mapping Convention of Return Value Types

Teradata uses the same mapping conventions for return value types that it uses for parameter types.

For a Java method that implements a scalar or aggregate UDF, the data type that you use for the return value maps to the SQL data type in the RETURNS clause of the CREATE FUNCTION or REPLACE FUNCTION statement. (Table UDFs do not have return values. The columns in the result rows that they produce are returned as output parameters.)

Consider the following CREATE FUNCTION statement for a scalar UDF:

```
CREATE FUNCTION factorial
  (x INTEGER)
  RETURNS INTEGER
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'JarUDF:UDFExample.fact';
```

The RETURNS clause specifies that the SQL data type of the return value is INTEGER. The signature of the *fact* method that implements the UDF looks like this:

```
public static int fact( int x ) { ... }
```

The default mapping convention is simple mapping, where SQL data types map to Java primitives. If no Java primitive can adequately map to an SQL type, then the default mapping convention is object mapping, where SQL data types map to Java classes.

Consider a factorial UDF that returns a DECIMAL value:

```
CREATE FUNCTION factorial
  (x INTEGER)
  RETURNS DECIMAL(8,2)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'JarUDF:UDFExample.fact';
```

Because the DECIMAL type does not map adequately to a Java primitive, the DECIMAL type maps to `java.math.BigDecimal`. The signature of the *fact* method that implements the UDF looks like this:

```
public static BigDecimal fact( int x ) { ... }
```

For details on how SQL data types map to Java data types, see [SQL Data Type Mapping](#).

## Overriding the Default Mapping of Return Value Types

For scalar or aggregate UDFs that can return a NULL result, simple mapping to Java primitives is not appropriate.

To override the default mapping, the EXTERNAL NAME clause in the CREATE FUNCTION or REPLACE FUNCTION statement must explicitly specify the Java data type of the return value.

For an example of how to override the default mapping of return value types, see [Example: Overriding Default Parameter Mapping to Handle NULLs](#).

## CLOB and BLOB Type Mapping

CLOB and BLOB SQL types as parameters or the return value of a CREATE FUNCTION or REPLACE FUNCTION statement map to java.sql.Clob and java.sql.Blob classes respectively.

The implementing class of java.sql.Clob is com.teradata.fnc.Clob and the implementing class of java.sql.Blob is com.teradata.fnc.Blob.

For details on the classes, including methods that you use to get a Blob or Clob for the result of a scalar or aggregate UDF, see [Java Application Classes](#)

## UDF Type Determines Additional Parameters

If the method implements an aggregate or table function, additional parameters are required.

IF you are writing ...	THEN use this syntax for the parameter list ...
a scalar function	<a href="#">Method Signature for Scalar UDFs</a>
an aggregate function	<a href="#">Method Signature for Aggregate UDFs</a>
a table function	<a href="#">Method Signature for Table UDFs</a>

## Scalar UDFs

### Method Signature for Scalar UDFs

#### Syntax

```
public class class_name {
    ...
    public static result_type method_name (
        [ type input_param [,...] ]
    )
}
```

```
{
  ...
}
```

## Syntax Elements

### ***class\_name***

Java class name.

### ***result\_type***

Java primitive or class corresponding to the SQL data type in the RETURNS clause of the corresponding CREATE FUNCTION statement.

### ***method\_name***

Java method name.

### ***type***

Data type of *input\_param*, a Java primitive or class corresponding to the SQL data type of the input argument.

### ***input\_param***

Input parameter for Java method. The method signature must have an *input\_param* for each parameter in the CREATE FUNCTION statement.

The maximum number of input parameters is 128.

## Example: Using Default Mapping of Parameter Types

Here is a code example that shows how to implement a Java method for a scalar UDF that maps an SQL INTEGER type parameter to the Java int primitive. This UDF does not handle NULL as an input argument, nor can it return NULL as a return value.

```
public class UDFExample {

    public static int fact( int x ) {
        if (x < 0)
            return 0;
        int factResult = 1;
        while (x > 1) {
            factResult = factResult * x;
            x = x - 1;
        }
    }
}
```

```

    }
    return factResult;
}

...

}

```

If the JAR file for the UDFExample class is called UDFExample.jar, the following statement registers UDFExample.jar and the *UDFExample* class with the database, and creates an SQL identifier called *JarUDF* for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!udfsrc/UDFExample.jar','JarUDF',0);
```

The corresponding CREATE FUNCTION statement to define the UDF looks like this :

```

CREATE FUNCTION factorial
  (x INTEGER)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'JarUDF:UDFExample.fact';

```

## Example: Overriding Default Parameter Mapping to Handle NULLs

Here is a code example that shows how to implement a Java method for a UDF that maps an SQL INTEGER type parameter to the java.lang.Integer class. This method can handle NULL as an input argument and return NULL as a return value.

```

public class UDFExample {

    public static Integer fact( Integer x ) {
        if (x == null)
            return null;
        int x_t = x.intValue();
        if (x_t < 0)
            return new Integer(0);
        int factResult = 1;
        while (x_t > 1) {
            factResult = factResult * x_t;

```

```

        x_t = x_t - 1;
    }
    return new Integer(factResult);
}

...

}

```

If the JAR file for the UDFExample class is called UDFExample.jar, the following statement registers UDFExample.jar and the *UDFExample* class with the database, and creates an SQL identifier called *JarUDF* for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!udfsrc/UDFExample.jar','JarUDF',0);
```

The corresponding CREATE FUNCTION statement to define the UDF looks like this :

```

CREATE FUNCTION factorial
  (x INTEGER)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
CALLED ON NULL INPUT
EXTERNAL NAME 'JarUDF:UDFExample.fact(java.lang.Integer) returns
java.lang.Integer';

```

## Related Information

For information on how a scalar UDF throws and handles exceptions, see [Exception Handling](#).

## Aggregate UDFs

When an SQL statement specifies an aggregate UDF, Vantage invokes the Java method that implements the aggregate UDF once for each item in the group, passing the detail values of a group through the input arguments.

In addition to the parameters that map to the input arguments of the UDF when it is specified in an SQL statement, the method signature for an aggregate UDF includes two special parameters for arguments that Vantage passes in. One of the arguments that Vantage passes in tells the method which aggregation phase it was invoked in so that the method knows how to combine the data passed in. Another argument provides a context for the method to retain intermediate results that it accumulates during the aggregation phases.



## Method Signature for Aggregate UDFs

### Syntax

```
import com.teradata.fnc.*;
import java.io.*;
import java.sql.*;

public class class_name {
    ...
    public static result_type method_name (
        Phase phase,
        Context[] context,
        [ input_parameter_specification [, ...] ]
    )
    {
        ...
    }
}
```

#### *input\_parameter\_specification*

```
type *input_parameter
```

### Syntax Elements

#### *result\_type*

a Java primitive or class corresponding to the SQL data type in the RETURNS clause of the corresponding CREATE FUNCTION statement.

#### *phase*

a required parameter that lets Vantage pass in the current aggregation phase. The aggregation phase determines how the method processes the data passed in. For details on the Phase class, see [com.teradata.fnc.Phase](#).

#### *context*

a required parameter that provides a way for the method to access intermediate storage to combine data passed in during the various aggregation phases. For details on the Context class, see [com.teradata.fnc.Context](#).

***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE FUNCTION definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is a Java primitive or class that corresponds to the SQL data type of *input\_parameter*.

## Basic Algorithm of an Aggregate UDF

Vantage invokes an aggregate UDF once for each item in an aggregation group, passing the detail values of a group through the input arguments. To accumulate summary information, the method must retain context each time it is called.

You do not need to understand or worry about how to create a group or how to code an aggregate UDF to deal with groups, because Teradata takes care of that. You only need to write the basic algorithm of combining the data passed to it to produce the desired result.

The basic algorithm consists of five possible phases of execution that an aggregate UDF goes through. To determine the phase of execution, and what actions the method must perform during each phase, call the `getPhase()` method on the *phase* input argument. The return value is a constant of the Phase class.

The following table describes the flow of an aggregate UDF. To see the flow of a window aggregate UDF, see [Window Aggregate UDFs](#).

Phase	Value of <i>phase.getPhase()</i>	Description
1	Phase.AGR_INIT	This is the first time Vantage is invoking the aggregate UDF for an aggregation group. The method must: <ul style="list-style-type: none"> <li>• Allocate intermediate storage and initialize it.</li> <li>• Process the first detail passed into the method.</li> </ul>
2	Phase.AGR_DETAIL	In this phase, Vantage calls the method once for each row to be aggregated for each group. The method must combine the UDF argument input data with the intermediate storage defined for the group.
3	Phase.AGR_COMBINE	This phase combines results from different AMPs for a specific group. The method must combine the data for the two intermediate storage areas passed in.
4	Phase.AGR_FINAL	No more input is expected for the group. The method must produce the final result for the group.
5	Phase.AGR_NODATA	This phase is only presented when there is absolutely no data to aggregate.

## Using a Switch Statement for the Basic Algorithm

One way to write an aggregate UDF to execute the proper code required for each aggregation phase is to use the Java switch statement.

To help illustrate what is required, the discussion uses code excerpts from a simple aggregate UDF that calculates the standard deviation. For the complete code example, see [UDF Code Examples](#)

Here is an example:

```
import com.teradata.fnc.*;
import java.io.*;
import java.sql.*;

...

public class UDFExample {

    public static Double stdDev(Phase phase, Context[] context, double x)
    throws SQLException
    {
        ...

        /* switch to determine the aggregation phase */
        switch (phase.getPhase()) {
            /* The AGR_INIT phase is executed once per group and          */
            /* allocates and initializes intermediate storage.             */
            case Phase.AGR_INIT:
                /* Get storage for intermediate aggregate values.          */
                ...
                /* Initialize the intermediate aggregate values.           */
                ...
                /* Fall through to the next phase because this phase      */
                /* passes in the first set of values for the group.         */
            /* The AGR_DETAIL phase is executed once for each selected     */
            /* row to aggregate. One copy will run on each AMP.            */
            case Phase.AGR_DETAIL:
                ...
                break;

            /* The AGR_COMBINE phase combines the results of              */
            /* ALL individual AMPs for each group.                         */
            case Phase.AGR_COMBINE:
                ...
        }
    }
}
```

```

        break;

        /* The AGR_FINAL phase returns the final result.          */
        /* It is called once for each group.                      */
        case Phase.AGR_FINAL:
            ...

        case Phase.AGR_NODATA:
            /* Not expecting no data. */
            return -1;

        default:
            /* If it gets here there must be an error because this */
            /* UDF does not accept any other phase options          */
            throw new SQLException("Invalid Phase", "38U05");
    }

    /* Save the intermediate results in the aggregate storage. */
    ...

    ...
}
}

```

## Related Information

FOR more information on ...	SEE ...
allocating and initializing intermediate storage in AGR_INIT phase	<a href="#">Allocating and Initializing Storage.</a>
accumulating intermediate results in the AGR_DETAIL phase	<a href="#">Saving Intermediate Results.</a>
combining intermediate storage areas from different AMPs	<a href="#">Combining Intermediate Storage Areas.</a>
returning the final group result	<a href="#">Producing the Final Group Result.</a>

## Window Aggregate UDFs

### Supported Window Types

You can apply a window specification to an aggregate function. The following window types are supported for aggregate UDFs:

Window Type	Aggregation Group	Partitioning Strategy
Reporting window	ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	Hash partitioning
Cumulative window	ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW or ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING	Hash partitioning
Moving window	<ul style="list-style-type: none"> <li>ROWS BETWEEN <i>value</i> PRECEDING AND CURRENT ROW</li> <li>ROWS BETWEEN CURRENT ROW AND <i>value</i> FOLLOWING</li> <li>ROWS BETWEEN <i>value</i> PRECEDING AND <i>value</i> FOLLOWING</li> <li>ROWS BETWEEN <i>value</i> PRECEDING AND <i>value</i> PRECEDING</li> <li>ROWS BETWEEN <i>value</i> FOLLOWING AND <i>value</i> FOLLOWING</li> </ul>	Hash partitioning and value partitioning

The following window types are *not* supported for aggregate UDFs:

Window Type	Aggregation Group
Moving window	<ul style="list-style-type: none"> <li>ROWS BETWEEN UNBOUNDED PRECEDING AND <i>value</i> FOLLOWING</li> <li>ROWS BETWEEN <i>value</i> PRECEDING AND UNBOUNDED FOLLOWING</li> </ul>

The partitioning strategy helps to avoid hot AMP situations where the values of the columns of the PARTITION BY clause result in the distribution of too many rows to the same partition or AMP. You should make sure that the method uses the right set of columns for the PARTITION BY clause to avoid potential skew situations for the reporting or cumulative aggregate cases. For more information, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

## Invoking a Window Aggregate UDF

Once an aggregate UDF is defined, you can invoke it using a partition window.

Consider the following aggregate UDF definition:

```

REPLACE FUNCTION myUdoa(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE(250)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA

```

```
EXTERNAL NAME 'INV:AggregateFunctions.myUdoa(com.teradata.fnc.Phase,
com.teradata.fnc.Context[],double) returns java.lang.Double';
```

You can invoke this aggregate UDF with the following query:

```
SELECT myUdoa(Product_ID) OVER() FROM tab_invf01102;
```

## Basic Algorithm of a Window Aggregate UDF

The following table describes the flow of a window aggregate UDF for a group.

Phase	Value of <i>phase.getPhase()</i>	Description
1	Phase.AGR_INIT	This phase is triggered once per partition at the start of a new aggregation group or first row. The method must: <ul style="list-style-type: none"> <li>• Allocate intermediate storage and initialize it.</li> <li>• Process the first detail passed into the method.</li> </ul>
2	Phase.AGR_DETAIL	This phase is triggered every time the forward row progresses. Vantage calls the method once for each row to be aggregated for each group. The method must combine the UDF argument input data with the intermediate storage defined for the group.
3	Phase.AGR_MOVINGTRAIL	This phase is applicable for the moving window type (noncumulative, nonreporting window type). This phase is triggered only by the last few rows of a moving window when the forward pointer to the window reaches the end of the group or end of the file. The phase does not provide any row or value to the method, but it is mainly used to indicate to the method that we are reaching the end of the group or file. The method can use this phase to adjust the necessary internal count or related values to reflect the actual size as the window diminishes towards the end of the group or file.
4	Phase.AGR_FINAL	This phase is invoked at the time the final evaluated result needs to be moved into the result row. No more input is expected for the group, and the method produces the final result for the group.
5	Phase.AGR_NODATA	This phase is only presented when there is absolutely no data to aggregate.

The Phase.AGR\_COMBINE phase is not applicable for window aggregate UDFs.

The following fields of FNC\_Context\_t are set up prior to invoking the window aggregate function:

- **pre\_window size:** This value is specified as part of the PRECEDING clause. The value of this field is negative if this points to a row that precedes the current row. This field is not applicable for the cumulative and reporting window types. It is initialized to zero in those cases.
- **post\_window size:** This value is specified as part of the FOLLOWING clause. The value of this field is negative if this points to a row that follows the current row. This field is not applicable for the cumulative and reporting window types. It is initialized to zero in those cases.
- **window\_size:** For the cumulative window type, this value is -1. For the reporting window type, the value is -2. For the regular window type, the value is set as post\_window size -pre\_window size +1 (+1 for the current row).

You must maintain a cache of rows corresponding to the window size. It may be useful to maintain a counter value that indicates the total rows read so far. For example, if C represents the counter, then the window of rows for evaluation would be:

```
if (C < window_size),
    window size would be C
else
    window size would be as indicated in function context.
```

The base of the window would be C - window\_size and the end of the window would correspond to C. The complexities of the various window combinations (PRECEDING/FOLLOWING/CURRENT, etc) is handled by Vantage. You only need to maintain the window cache of rows and implement the semantics of the function on this cache of rows.

To see sample Java code that implements a window aggregate function, see [Java Window Aggregate Function](#).

## Intermediate Aggregate Storage

All aggregate UDFs require intermediate storage to combine data passed in during the various aggregation phases.

This topic describes how an aggregate UDF handles intermediate storage during each aggregation phase. To help illustrate what is required, the discussion uses code excerpts from a simple aggregate UDF that calculates the arithmetic sum. For the complete code example, see [UDF Code Examples](#)

## Context[] Parameter

As discussed in [Aggregate UDFs](#), the parameter list includes an array of com.teradata.fnc.Context objects.

The Context class provides methods that are useful for aggregate functions.

Method	Description
initCtx()	Allocates and initializes the aggregate intermediate storage area. An aggregate UDF uses this method in the Phase.AGR_INIT phase of aggregation.

Method	Description
getBytes()	Returns data that was stored in one of two intermediate aggregate storage areas.
getObject()	
setBytes()	Stores intermediate results specified by obj into the aggregate storage area.
setObject()	

## Defining What to Store in the Storage Area

The type of intermediate results that an aggregate UDF needs to save depends on the type of calculation that the UDF performs and whether the method wants to represent the intermediate results as a byte array or as an object.

Consider a standard deviation function `STD_DEV(x)` that uses the following equation:

$$s = \sqrt{\frac{\sum X^2}{N} - \left(\frac{\sum X}{N}\right)^2}$$

Based on the calculation, the UDF needs to store the following:

- `N`
- `sum(X2)`
- `sum(X)`

Here is a Java class called `agr_storage` with fields that match the necessary intermediate values:

```
class agr_storage implements Serializable{
    double count;
    double x_sq;
    double x_sum;

    public agr_storage(double a, double b, double c){
        count = a;
        x_sq = b;
        x_sum = c;
    }
}
```

Alternatively, especially when better performance is required, the method that implements the aggregate UDF can use a `ByteBuffer` for the intermediate results and use the `putDouble()` and `getDouble()` methods on the `ByteBuffer` for the necessary intermediate values.



## Allocating and Initializing Storage

During the Phase.AGR\_INIT aggregation phase, a UDF must call the `initCtx()` method on the *context* input argument to allocate and initialize the intermediate storage area based on the object or byte array that the method needs to store.

The UDF cannot request more memory than the value of *interim\_size* in the CLASS AGGREGATE clause of the CREATE FUNCTION statement. For best performance, allocate only enough memory to satisfy the needs of the UDF for intermediate storage.

If the method stores a byte array in the intermediate storage area, the size of the intermediate storage area can be determined by the total number of bytes required. For the standard deviation example, the total number of bytes required is 24, enough for 3 double values.

Alternatively, if the method stores an object in the intermediate storage area, the memory required can be computed as follows.

```
public static int getSize(Object obj){
    int size=0;
    try{
        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutput s = new ObjectOutputStream(barr);
        s.writeObject(obj);
        s.close();
        size=barr.toByteArray().length;
        System.out.println("obj="+obj+",size="+size);

    }catch(IOException e){
        e.printStackTrace();
    }
    return size;
}
```

For the standard deviation example, the following code initializes an `agr_storage` object and allocates intermediate storage for the object:

```
agr_storage s1 = new agr_storage(0,0,0);
context[0].initCtx(s1);
```

Alternatively, for a method that uses a byte array to store intermediate results, the code looks like this:

```
context[0].initCtx(24);
```

## Saving Intermediate Results

During the Phase.AGR\_INIT aggregation phase, a UDF must also save the first set of data values passed in through arguments into the intermediate aggregate storage area. The first set of data values are for the first row to aggregate for the group.

Thereafter, each time the method is invoked during the Phase.AGR\_DETAIL aggregation phase, it must accumulate the row data passed in through arguments into the intermediate aggregate storage area for the specific group. Each group being aggregated to has a separate intermediate storage area.

During the Phase.AGR\_DETAIL aggregation phase, the method gains access to previously accumulated data by calling getObject() or getBytes() on the *context* input argument. For the standard deviation example, the following code gains access to previously accumulated data:

```
agr_storage s1 = (agr_storage)context[0].getObject(1);
```

Alternatively, if the method used a byte array to store data, the following code gains access to the previously stored values:

```
ByteBuffer s1 = ByteBuffer.wrap(context[0].getBytes(1));
double count = s1.getDouble();
double x_sq = s1.getDouble();
double x_sum = s1.getDouble();
```

In the standard deviation example, the x input argument is the column the standard deviation is being calculated for. For the standard deviation calculation, the method uses the value of x to calculate the following:

- $\text{sum}(X^2)$
- $\text{sum}(X)$

If the method uses the agr\_storage object to store data, the following code performs the necessary calculations using the column value, and then combine the results with the intermediate storage:

```
s1.count++;
s1.x_sq += x*x;
s1.x_sum += x;
context[0].setObject( 1, s1 );
```

Alternatively, if the method uses a byte array to store data, the code looks like this:

```
s1 = ByteBuffer.allocate(24);
count++;
x_sq += x*x;
x_sum += x;
```

```
s1.putDouble(count);
s1.putDouble(x_sq);
s1.putDouble(x_sum);
context[0].setBytes( 1, s1.array() );
```

## Combining Intermediate Storage Areas

During the AGR\_COMBINE aggregation phase, a UDF must combine two intermediate aggregate storage areas into one storage area for each group that is being aggregated. The storage areas that the UDF is combining are the results from different AMPs for a specific group.

The ultimate result is to create one summary aggregate storage area for each group on which the aggregate UDF operates.

In the standard deviation example, the method can use another `agr_storage` instance for the data in the second intermediate aggregate storage area:

```
agr_storage s2 = (agr_storage)context[0].getObject(2);
```

The following statements combine the aggregate storage areas:

```
s1.count += s2.count;
s1.x_sq += s2.x_sq;
s1.x_sum += s2.x_sum;
```

Alternatively, if the method uses a byte array for the data in the intermediate aggregate storage area, the code looks like this:

```
ByteBuffer s2 = ByteBuffer.wrap(context[0].getBytes(2));
count += s2.getDouble();
x_sq += s2.getDouble();
x_sum += s2.getDouble();
```

---

### Note:

The AGR\_COMBINE aggregation phase is not applicable for window aggregate UDFs.

---

## Producing the Final Group Result

During the AGR\_FINAL aggregation phase, a UDF produces the final result for the group. The method must set the result value to the final value.

In the standard deviation example, the method can use the following code to calculate the final value:

```
double term2 = s1.x_sum/s1.count;
double variance = s1.x_sq/s1.count - term2*term2;
```

The following statement sets the result value to the final value of the standard deviation:

```
return new Double(Math.sqrt(variance));
```

## Aggregate Cache and Aggregate UDFs

To enhance performance, Vantage caches intermediate aggregate storage areas. The size of the aggregate cache is fixed at 1 MB.

If the number of intermediate storage areas exceeds the capacity of the aggregate cache, Vantage pages the least recently used intermediate storage areas to a spool file on disk, which causes a larger number of aggregation phases to occur and impacts performance.

For details on improving aggregate cache usage for an aggregate UDF, including how to determine the maximum number of intermediate storage areas, see the information about CREATE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Related Information

Related information appears in other documents and in other topics of this document.

FOR more information on ...	SEE ...
the Phase and Context classes	<a href="#">Java Application Classes</a> .
example code that shows each aggregation phase	<a href="#">UDF Code Examples</a> .
optimizing the aggregate cache	CREATE FUNCTION information in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

## Table UDFs

A table function is invoked in the FROM clause of an SQL SELECT statement and returns a table a row at a time in a loop to the SELECT statement. The function can produce the rows of a table from the input arguments passed to it or by reading an external file or message queue.

Like a method that implements a scalar or aggregate UDF, the parameter list for a method that implements a table UDF includes input parameters that map to the input arguments of the UDF when it is specified in an SQL statement.

Unlike a method that implements a scalar or aggregate UDF, the parameter list of a method that implements a table UDF includes output parameters that map to the columns in the result rows that the table UDF produces. Another difference is that the return type for a method that implements a table UDF is always *void*.

A table function can have:

- 128 input parameters
- As many output parameters as defined by the RETURNS TABLE clause of the CREATE FUNCTION or REPLACE FUNCTION statement for the table function.

The number of output parameters is limited by the maximum number of columns that can be defined for a regular table.

## Teradata Application Classes

Teradata provides several Java application classes that you use to implement table UDFs.

Class	Description
com.teradata.fnc.AmpInfo	Defines methods that return AMP-specific information that a local copy of a table UDF can use to configure itself to use the correct resources.
com.teradata.fnc.NodeInfo	Defines methods that return node IDs and AMP IDs for all online AMP vprocs, allowing table UDFs to configure themselves to run on specific AMPs.
com.teradata.fnc.Tbl	Provides methods for Java table UDF processing.

For details, see [Java Application Classes](#)

## Fixed Result Row Specification

If the CREATE FUNCTION or REPLACE FUNCTION statement specifies a column list for the table function row result, then the table function is said to have a fixed result row specification. The number of output parameters in the Java method parameter list matches the number of columns in the column list in the RETURNS TABLE clause of the CREATE FUNCTION or REPLACE FUNCTION statement.

Here is an example of a CREATE FUNCTION statement for a table function with a fixed result row specification:

```
CREATE FUNCTION getStoreData (FileToRead INTEGER)
RETURNS TABLE (StoreNo INTEGER, ItemNo INTEGER, QuantSold INTEGER)
LANGUAGE JAVA
NO SQL
EXTERNAL NAME 'JarUDF:UDFExample.getStoreData'
PARAMETER STYLE JAVA;
```

In this example, the number of result parameters in the table UDF parameter list is three.

## Dynamic Result Row Specification

If the CREATE FUNCTION or REPLACE FUNCTION statement specifies a RETURNS TABLE VARYING COLUMNS clause, then the table function is said to have a dynamic result row specification. The number of output parameters in the Java method parameter list matches the maximum number of columns in the RETURNS TABLE VARYING COLUMNS clause of the CREATE FUNCTION or REPLACE FUNCTION statement.

The actual number and data types of the output parameters that the method needs to return values in is specified dynamically at runtime in the SELECT statement that invokes the table function.

Here is an example of a CREATE FUNCTION statement for a table function with a dynamic result row specification:

```
CREATE FUNCTION getStoreData (FileToRead INTEGER)
RETURNS TABLE VARYING COLUMNS (10)
LANGUAGE JAVA
NO SQL
EXTERNAL NAME 'JarUDF:UDFExample.getStoreData'
PARAMETER STYLE JAVA;
```

In this example, the number of output parameters in the table UDF parameter list is 10.

## Method Signature for Table UDFs

### Syntax

```
public class class_name {
    ...
    public static void method_name (
        [ type input_param [...] ]
        result_type result [...]
    )
    {
        ...
    }
    ...
}
```

### Syntax Elements

***class\_name***

Java class name.

***method\_name***

Java method name.

***type***

Data type of *input\_param*, a Java primitive or class corresponding to the SQL data type of the input argument.

***input\_param***

Input parameter for Java method. The signature must have an *input\_param* for each parameter in the CREATE FUNCTION statement.

The maximum number of input parameters is 128.

***result\_type***

Java primitive or class corresponding to the SQL data type in the RETURNS clause of the corresponding CREATE FUNCTION statement.

If the table UDF is defined with fixed result row specification, *result\_type* is a Java primitive or class that matches the SQL data type of the corresponding column in the RETURNS TABLE clause of the CREATE FUNCTION statement.

If the table UDF is defined with dynamic result row specification, *result\_type* is `java.lang.Object[]`, because the actual data types of the result row arguments are unknown until function invocation.

***result***

Row result output parameter, determined by the corresponding CREATE FUNCTION or REPLACE FUNCTION definition.

If the table UDF is defined with fixed result row specification, the method signature has a *result* for each column in the RETURNS TABLE clause in the corresponding CREATE FUNCTION definition.

If the table UDF is defined with dynamic result row specification, the method signature has the maximum number of columns in the RETURNS TABLE VARYING COLUMNS clause in the corresponding CREATE FUNCTION definition. During execution, the Java method can invoke the `Tbl.getColDef()` method to get the actual result types.

The method must return at least one result row output argument.

## Example: Table UDF with Fixed Result Row Specification

Here is a code excerpt that shows the method signature for a method that implements a table UDF:

```

public class UDFExample {

    ...

    public static void getStoreData( int    storedata, /* input arg */
                                     int[] storeno,   /* output arg */
                                     int[] itemno,    /* output arg */
                                     int[] quantsold) /* output arg */
    {
        ...
    }

    ...

}

```

If the JAR file for the UDFExample class is called UDFExample.jar, the following statement registers UDFExample.jar and the *UDFExample* class with the database, and creates an SQL identifier called *JarUDF* for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!udfsrc/UDFExample.jar','JarUDF',0);
```

The corresponding CREATE FUNCTION statement to define the UDF looks like this :

```

CREATE FUNCTION getStoreData
  (FileToRead INTEGER)
RETURNS TABLE
  (StoreNo    INTEGER
   ,ItemNo    INTEGER
   ,QuantSold INTEGER)
LANGUAGE JAVA
NO SQL
EXTERNAL NAME 'JarUDF:UDFExample.getStoreData'
PARAMETER STYLE JAVA;

```

Here is an example of an INSERT ... SELECT statement that invokes the table function in the FROM clause:

```

INSERT INTO Sales_Table
SELECT *
FROM TABLE (getStoreData(9005)) AS tf;

```



## Example: Table UDF with Dynamic Result Row Specification

Here is a code excerpt that shows the method signature for a method that implements a table UDF with dynamic result row specification:

```
public class UDFExample {
    ...
    public static void getStoreData( int      store, /* input arg */
                                     java.lang.Object[] c1, /* output arg */
                                     java.lang.Object[] c2, /* output arg */
                                     java.lang.Object[] c3, /* output arg */
                                     java.lang.Object[] c4) /* output arg */
    {
        ...
    }
    ...
}
```

If the JAR file for the UDFExample class is called UDFExample.jar, the following statement registers UDFExample.jar and the *UDFExample* class with the database, and creates an SQL identifier called *JarUDF* for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!udfsrc/UDFExample.jar','JarUDF',0);
```

The corresponding CREATE FUNCTION statement to define the UDF looks like this:

```
CREATE FUNCTION getStoreData
  (FileToRead INTEGER)
RETURNS TABLE VARYING COLUMNS (4)
LANGUAGE Java
NO SQL
EXTERNAL NAME 'JarUDF:UDFExample.getStoreData'
PARAMETER STYLE Java;
```

Here is an example of an INSERT ... SELECT statement that invokes the table function in the FROM clause:

```
INSERT INTO Sales_Table
SELECT *
FROM TABLE (getStoreData(9005)
  RETURNS (Store INTEGER, Item INTEGER, Quantity INTEGER)) AS tf;
```

## Constant Mode Table UDFs

This section provides guidelines on how to write a Java method for a table UDF that can handle an invocation from a SELECT statement that specifies constant expression input arguments.

For example, the following statement invokes `table_function_1` with a constant argument:

```
SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;
```

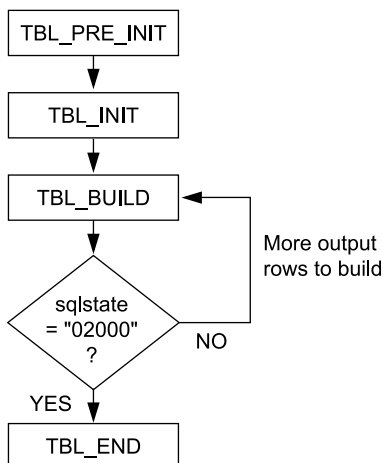
### Note:

Although this section discusses how to implement a table function that only handles constant expression input arguments, you can design a table function that can handle constant and variable input arguments. For details on how to implement a variable mode table function, see [Variable Mode Table UDFs](#).

## Phases of a Constant Mode Table UDF

Vantage repeatedly invokes a Java method that implements a constant mode table UDF, allowing the method to pass through different phases of execution where it can perform initialization, access system resources, and build output rows. The method uses `com.Teradata.fnc.Tbl.getPhase()` to determine the phase in which it was called and what action to take.

Here are the phases that the function passes through:



For details on implementing each phase of a constant mode table function, see [Implementation Guidelines](#).

## Implementation Guidelines

Here are the basic steps you take to write a Java method for a table UDF that is invoked with constant expression input arguments:

1. Define the parameter list in the order that the CREATE FUNCTION statement specifies the parameters.  
For more information, see [Method Signature for Table UDFs](#).
2. If the table UDF is defined with dynamic result row specification, call `Tbl.getColDef()` to get the actual number and data types of the result row arguments that the method must return.
3. Call `Tbl.getPhase()` and verify that the mode (the return value) is `Tbl.TBL_MODE_CONST`, indicating that the table function was invoked with constant expression input arguments.
4. Use the value that `Tbl.getPhase()` returns in the *phase* argument to determine the phase in which Vantage invoked the method and what action to take.

IF the value is...	THEN the method...
<code>Tbl.TBL_PRE_INIT</code>	<p>must decide whether it should be the controlling copy for the table UDF (other copies of the method are executed on other AMP vprocs).</p> <p>If the method wants to provide control context to all other copies of the method, the method must call <code>Tbl.control()</code>.</p> <p>If the method does not want to be the controlling copy for the table UDF, or if the method is designed without the need for a controlling method, the method can simply return and do nothing during this phase.</p> <p>All copies of the method must complete this phase before any copy continues to the <code>Tbl.TBL_INIT</code> phase.</p>
<code>Tbl.TBL_INIT</code>	<p>can start to access system resources, such as local files, if there is a need to do so.</p> <p>Any copy of the method that does not want to participate further must call <code>Tbl.optOut()</code>. Vantage will not call the method again.</p> <p>All copies of the method must complete this phase before any copy continues to the <code>Tbl.TBL_BUILD</code> phase.</p>
<code>Tbl.TBL_BUILD</code>	<p>should take one of the following actions:</p> <p>If the method has...</p> <ul style="list-style-type: none"> <li>• a row to build, then build an output row by filling out each output argument that has a non-null input value. The method remains in the <code>Tbl.TBL_BUILD</code> phase.</li> <li>• no row to build, then throw an <code>SQLException</code>, setting the <code>SQLState</code> field to "02000" to indicate no data. The method continues to the <code>Tbl.TBL_END</code> phase.</li> </ul>
<code>Tbl.TBL_END</code>	<p>(if it is the the controlling copy of the table UDF) is called with this phase after all other copies of the table UDF have completed this phase, allowing the controlling function to do any notification to the external world.</p> <p>Vantage does not invoke the method again after it returns from this phase.</p>

IF the value is...	THEN the method...
Tbl.TBL_ABORT	is being aborted. A method can be called at any time with this phase, which is only entered when a copy of the table functions calls Tbl.abort(). This phase is not entered when the method aborts for an external reason, such as a user abort.

5. If the method detects an error, throw an SQLException. For more information, see [Exception Handling](#).

## Design Considerations

A method that implements a constant mode table UDF is executed on all AMP vprocs and each copy is passed the same input arguments. Each copy is called repeatedly until the method indicates it is finished.

During the design of a table UDF, you must determine whether it makes sense for all methods on all AMP vprocs to participate in the processing.

For example, a typical constant mode table UDF most likely reads data from outside the database to produce result rows. If the external data is only available on one node, it might be practical to have only one method copy on one vproc do anything useful. On the other hand, if the external data is available on each node, then perhaps it can be read from all AMP vprocs.

IF you want ...	THEN ...
all copies of the method to participate in the processing	<p>use the following code excerpt as a guideline to implement your method.</p> <pre> Tbl tbl = new Tbl(); int[] phase = new int[1];  if ( tbl.getPhase(phase) != Tbl.TBL_MODE_CONST ) {     /* set SQLSTATE to an error and return */     throw new SQLException("Wrong mode", "38U06");     return; }  /* depending on the phase decide what to do */ switch(phase[0]) {     case Tbl.TBL_PRE_INIT:     {         break;     }     case Tbl.TBL_INIT:     {         /* Perform preprocessing or initializations here. */         ...         break;     }     case Tbl.TBL_BUILD:     {         /* Read from files and build the result row here.      */         /* On EOF, set SQLSTATE to "02000" (no row to build). */ </pre>

IF you want ...	THEN ...
	<pre>         ...         break;     }     case Tbl.TBL_END:     {         /* Everyone done. */         ...         break;     }     case Tbl.TBL_ABORT:     {         /* A copy called Tbl.abort(). */         ...         break;     } } </pre>
<p>a method that that can run on any AMP and only needs one copy to participate</p>	<p>call <code>Tbl.firstParticipant()</code> from all copies of the method. The first copy to make the call is the copy that participates. All other copies must call <code>Tbl.optOut()</code> and return. Use the following code excerpt as a guideline to implement your method:</p> <pre> Tbl tbl = new Tbl(); int[] phase = new int[1];  if ( tbl.getPhase(phase) != Tbl.TBL_MODE_CONST ) {     /* set SQLSTATE to an error and return */     throw new SQLException("Wrong mode", "38U06");     return; }  /* depending on the phase decide what to do */ switch(phase[0]) {     case Tbl.TBL_PRE_INIT:     {         if (tbl.firstParticipant()) {             return; /* participant */         } else {             if (!tbl.optOut()) { /* not a participant */                 throw new SQLException("Opt out failure", "38U06");             }             return;         }         break;     }     case Tbl.TBL_INIT:     {         /* Perform preprocessing or initializations here. */         ...         break;     } } </pre>

IF you want ...	THEN ...
	<pre> case Tbl.TBL_BUILD: {     /* Read from files and build the result row here.    */     /* On EOF, set SQLSTATE to "02000" (no row to build). */     ...     break; } case Tbl.TBL_END: {     /* Everything is done. */     ...     break; } case Tbl.TBL_ABORT: {     /* A copy called Tbl.abort(). */     ...     break; } } </pre>
<p>one copy of the method to be the controlling copy of all other copies running on all other AMP vprocs</p>	<p>call Tbl.control() to designate a copy of the table UDF as the controlling copy. Distribute data to other copies by calling Tbl.setCtrlCtx(). Use the following code excerpt as a guideline to implement your method.</p> <pre> class ctrl_ctx implements Serializable {     int ctrl_AMP;     int qfd;     ... } ;  public class UDFExample {      public static void getStoreData(int storeData,                                     int[] storeNo,                                     int[] itemNo)      {         ctrl_ctx options;         Tbl tbl = new Tbl();         int[] phase = new int[1];         if ( tbl.getPhase(phase) != Tbl.TBL_MODE_CONST )         {             /* set SQLSTATE to an error and return */             throw new SQLException("Wrong mode", "38U06");             return;         }         /* Depending on the phase decide what to do. */         switch(phase[0])         {             case Tbl.TBL_PRE_INIT:                 AMPInfo localCfg = new AMPInfo();                 /* Run controlling copy on lowest AMP on node. */ </pre>

IF you want ...	THEN ...
	<pre>         if (localCfg.lowestAMPOnNode())         {             /* Run on node that can access external file. */             ...              if ( tbl.control() )             {                 /* Use scratchpad to distribute data to */                 /* copies during the Tbl.TBL_INIT phase. */                 options.ctrl_AMP = localCfg.getAMPId();                 tbl.setCtrlCtx(options);                 ...             }         }         break;     case Tbl.TBL_INIT:         /* Get the data from the controlling copy. */         options = tbl.getCtrlCtx();         ...         break;     case Tbl.TBL_BUILD:         /* Build result row or set SQLSTATE */         /* to "02000" if no data. */         ...         break;     case Tbl.TBL_END:         /* Everyone done. */         ...         break;     case Tbl.TBL_ABORT:         /* A copy called Tbl.abort(). */         ...         break;     } } </pre>

## Variable Mode Table UDFs

This section provides guidelines on how to write a Java method for a table UDF that can handle an invocation where the SELECT statement specifies columns from a derived table as input arguments. For example, the following statement invokes `table_function_2` using `column_1` from the preceding derived table with the `t1` correlation name:

```

SELECT *
FROM ( SELECT column_1
      FROM table_1
      WHERE column_2 > 65 ) AS t1,

```

```
TABLE (table_function_2(t1.column_1)) AS t2 (c1, c2, c3)
WHERE t1.column_3 = t2.c3;
```

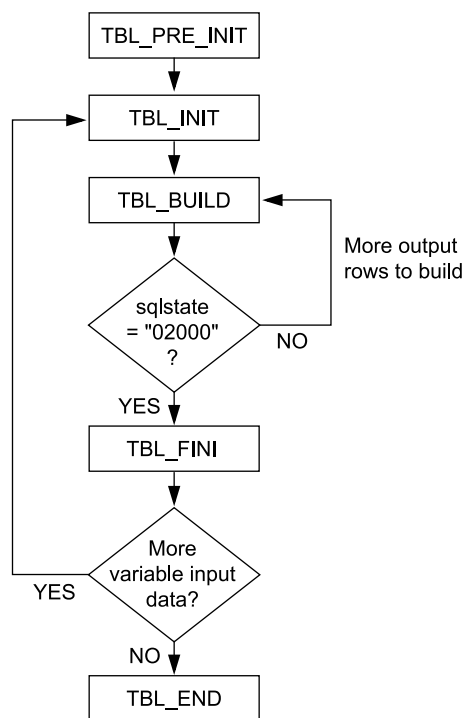
**Note:**

Although this section discusses how to implement a method that only handles variable input arguments, you can write a method that can handle constant and variable input arguments. For details on constant mode table UDFs, see [Constant Mode Table UDFs](#).

## Phases of a Variable Mode Table UDF

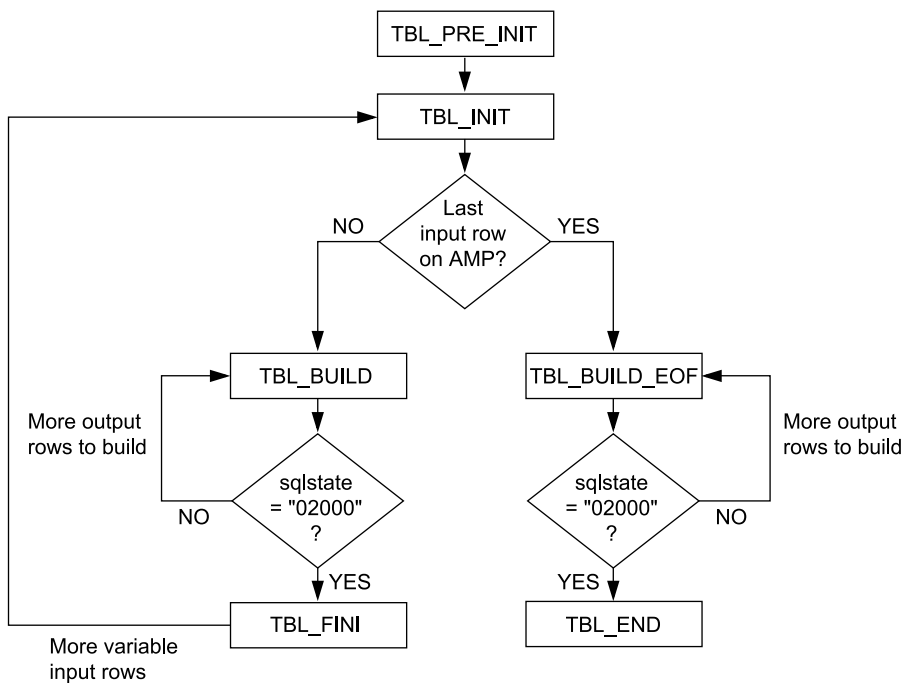
Vantage repeatedly invokes the Java method for a variable mode table UDF, allowing the method to pass through different phases where it initializes itself, accesses system resources, and builds output rows. The method uses `Tbl.getPhase()` or `Tbl.getPhaseEx()` to determine which phase it is in and what action to take.

A method that uses `Tbl.getPhase()` passes through the following phases:



Alternatively, a table UDF that needs to know when it is passed in the last qualified row on an AMP (perhaps because the UDF does not return a row until after it processes all of the input rows) can use `Tbl.getPhaseEx()` with the `TBL_LASTROW` option and pass through the following phases:





For details on implementing each phase of a variable mode table UDF, see [Implementation Guidelines](#).

## Implementation Guidelines

Here are the basic steps you take to write a Java method that implements a table UDF that is invoked using the columns from a derived table as input arguments:

1. Define the parameter list in the order that the `CREATE FUNCTION` statement specifies the parameters.  
For more information, see [Method Signature for Table UDFs](#).
2. If the table UDF is defined with dynamic result row specification, call `Tbl.getColDef()` to get the actual number and data types of the result row arguments that the method must return.
3. Call `Tbl.getPhase()` or `Tbl.getPhaseEx()` and verify that the return value is `Tbl.TBL_MODE_VARY`, indicating that the table UDF was invoked with the columns from a derived table as input arguments.
4. Use the value that `Tbl.getPhase()` or `Tbl.getPhaseEx()` returns in the *phase* argument to determine the phase in which Vantage invoked the method and what action to take.

IF the value is ...	THEN the method ...
<code>Tbl.TBL_PRE_INIT</code>	<p>is being called for the first time for all the rows that it will be called for. The input arguments to the method contain the first set of data.</p> <p>During this phase, the method has an opportunity to establish overall global context, but should not build any result row.</p> <p>The method continues to the <code>TBL_INIT</code> phase.</p>

IF the value is ...	THEN the method ...
Tbl.TBL_INIT	<p>can call Tbl.allocCtx() to initialize the function context to use as a scratchpad for data between local iterations of the method, if there is a need to do so.</p> <p>The input arguments to the method contain the first set of data.</p> <p>During this phase, the method should not build any result row.</p> <p>The method continues to the TBL_BUILD phase.</p>
Tbl.TBL_BUILD	<p>should take one of the following actions.</p> <ul style="list-style-type: none"> <li>• If the method has a row to build, then build an output row by filling out each output argument that has a non-null input value. The method remains in the TBL_BUILD phase.</li> <li>• If the method has no row to build, then throw an SQLException, setting the SQLState field to "02000" to indicate no data. The method continues to the TBL_FINI phase.</li> </ul> <p>If using Tbl.getPhaseEx(), the following actions are done depending on the option specified:</p> <ul style="list-style-type: none"> <li>• If the TBL_NEWROW option is set, then call the function with a new row with a phase of TBL_BUILD.</li> <li>• If the TBL_NEWROWEOF option and EOF are set, then call the table function with a new row with a phase of TBL_BUILD.</li> <li>• If the TBL_LASTROW option is set, the function remains in the TBL_BUILD phase until it is passed in the last set of data, where it continues to the TBL_BUILD_EOF phase.</li> </ul>
Tbl.TBL_BUILD_EOF (only getPhaseEx() can return this value)	<p>is being called after the last input row on the AMP was passed in. The UDF has an opportunity to output a summary row of what it has collected in memory during previous calls when the phase was TBL_BUILD. The method should take one of the following actions.</p> <ul style="list-style-type: none"> <li>• If the method has a row to build, then build an output row by filling out each output argument that has a non-null input value. The method remains in the Tbl.TBL_BUILD_EOF phase.</li> <li>• If the method has no row to build, then throw an SQLException, setting the SQLState field to "02000" to indicate no data. The method continues to the Tbl.TBL_END phase.</li> </ul>
Tbl.TBL_FINI	<p>can perform any initialization, such as clearing the scratchpad, in preparation for the possible next set of data.</p> <p>If there is more variable input data, the method returns to the Tbl.TBL_INIT phase. Otherwise, the method continues to the TBL_END phase.</p>
Tbl.TBL_END	<p>can do any required notification to the external world that processing is complete. Vantage does not invoke the method again after it returns from this phase.</p>
Tbl.TBL_ABORT	<p>is being aborted. A method can be called at any time with this phase, which is only entered when a copy of the table functions calls Tbl.abort(). This phase is not entered when the method aborts for an external reason, such as a user abort.</p>

5. If the method detects an error, throw an SQLException. For more information, see [Exception Handling](#).

In general, you should understand the following row states:

- When the first row is passed, during the TBL\_PRE\_INIT phase.

- When a row is being returned, as indicated by the value of the SQLState field in the TBL\_BUILD phase.
- When you want a new row. You can define this using the TBL\_NEWROW or TBL\_NEWROWEOF options of Tbl.getPhaseEx().
- When the end of file has been encountered. You can determine this using the TBL\_LASTROW option of Tbl.getPhaseEx().

## Improving Performance with Phase Reductions

The Tbl.getPhaseEx() options give you more control of table phase transitions when developing variable mode table functions. You can use them to reduce the number of phase transitions required during execution of a table function, thus reducing the number of UDF invocations and improving table function performance.

The following table compares the number of phase transitions required for each row if you do not use the Tbl.getPhaseEx() options and if you do use the various options. The phases are P (TBL\_PRE\_INIT), I (TBL\_INIT), B (TBL\_BUILD), B EOF (TBL\_BUILD signalling EOF, not TBL\_BUILD\_EOF), F (TBL\_FINI), and E (TBL\_END). X is the scale factor of input to output rows.

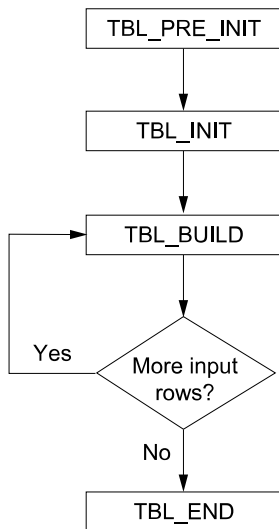
Processing Mode	Required phases if the Tbl.getPhaseEx() options are not used	Required phases if the specified Tbl.getPhaseEx() options are used
1:1 (one row in : one row out)	I, B, B EOF, F	B, if the TBL_NEWROW option is set.
1:M (one row in : many rows out)	I, B*X, B EOF, F	B * X, if the TBL_NEWROWEOF option is set.
M:1 (many rows in : one row out)	I, B EOF, F	B * X, if the TBL_NEWROW   TBL_LASTROW options are set.

For example:

In a 1:1 processing mode, use the TBL\_NEWROW option to get a new row on every TBL\_BUILD call.

```
tbl.getPhaseEx(phase, tbl.TBL_NEWROW);
```

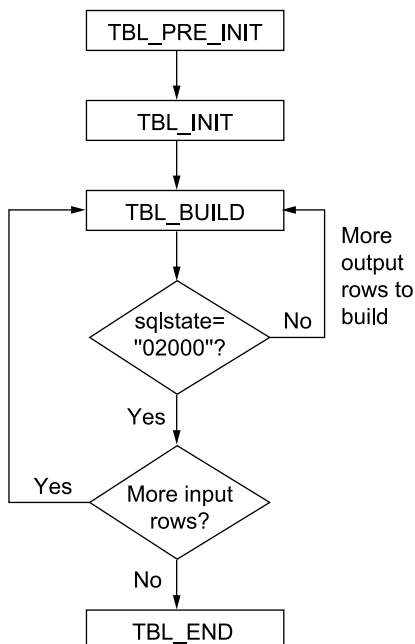
In this case, the function passes through the following phases:



In a 1:M processing mode, use the TBL\_NEWROWEOF option to get a new row when EOF is signaled.

```
tbl.getPhaseEx(phase, tbl.TBL_NEWROWEOF);
```

In this case, the function passes through the following phases:



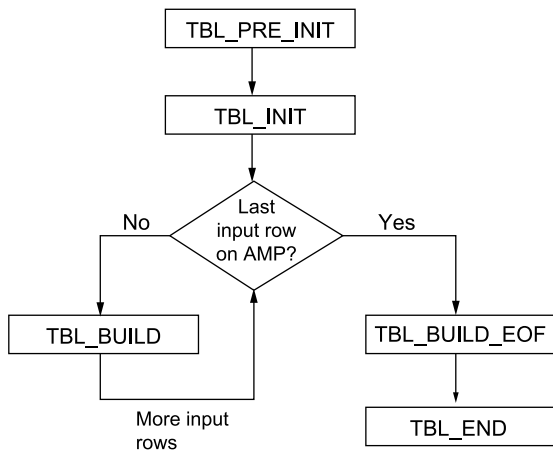
In a M:1 processing mode, you can use:

```
tbl.getPhaseEx(phase, tbl.TBL_LASTROW | tbl.TBL_NEWROW);
```

In a combined M:1 and 1:M processing mode, you can use the following that does not pass a new row until EOF:

```
tbl.getPhaseEx(phase, tbl.TBL_LASTROW |tbl.TBL_NEWROWEOF);
```

In this case, the function passes through the following phases:



For more information, see [getPhaseEx\(int\[\] phase, int option\)](#).

## Design Considerations

The AMP vprocs that participate are those containing rows pertaining to the input data provided to the method from the correlated table name specified prior to the table function in the SELECT statement. The assumption is that the input argument determines what to process. Each copy of the method will be called with the same input data repeatedly until the method returns with the no more data condition.

Although it might be possible to read additional data from outside the database in this mode, it probably will not work. This is because the participating AMP vprocs are determined based on where the selected database rows reside. It might not be on all AMP vprocs.

Use the following code excerpt as a guideline to implement your method using `Tbl.getPhase()`:

```

public class local_ctx {
    String xml_ctx;
    public void local_ctx() {
        xml_ctx = new String(50);
    }
} ;

public class UDFExample {

```

```

public static void getStoreData( int storeData,
                                int[] storeNo,
                                int[] itemNo)
{
    Tbl tbl = new Tbl();
    int[] phase = new int[1];
    local_ctx state_info;
    if ( tbl.getPhase(phase) != Tbl.TBL_MODE_VARY )
    {
        /* set SQLSTATE to an error and return */
        throw new SQLException("Wrong mode", "38U06");
        return;
    }

    /* depending on the phase decide what to do */
    switch(phase[0])
    {
        case Tbl.TBL_PRE_INIT:
            state_info = new local_ctx();
            ...
            break;
        case Tbl.TBL_INIT:
            /* Allocate scratchpad to retain data between iterations. */
            tbl.allocCtx(state_info);
            /* Preprocess data here. */
            ...
            break;
        case Tbl.TBL_BUILD:
            /* Get scratchpad and build the result row here. */
            state_info = (local_ctx)tbl.getCtxObject();
            ...
            /* Or, if no more rows to build, set SQLSTATE to "02000". */
            throw new SQLException("no more data", "02000");
            ...
            break;
        case Tbl.TBL_FINI:
            /* Reset for the next set of data. */
            ...
            break;
        case Tbl.TBL_END:
            /* Everyone done. */
            ...
            break;
    }
}

```

```

    }
}

```

If your table UDF does not build a result row until after it receives all of the input rows for the AMP on which it runs, you can use `Tbl.getPhaseEx()` with the `TBL_LASTROW` option instead of `Tbl.getPhase()`. This will build the row during the `TBL_BUILD_EOF` phase. Use the following code excerpt as a guideline to implement your UDF using `Tbl.getPhaseEx()`:

```

public class local_ctx {
    String xml_ctx;
    public void local_ctx() {
        xml_ctx = new String(50);
    }
} ;

public class UDFExample {

    public static void getStoreData( int storeData,
                                     int[] storeNo,
                                     int[] itemNo)

    {
        Tbl tbl = new Tbl();
        int[] phase = new int[1];
        local_ctx state_info;
        if ( tbl.getPhaseEx(phase, Tbl.TBL_LASTROW) != Tbl.TBL_MODE_VARY )
        {
            /* set SQLSTATE to an error and return */
            throw new SQLException("Wrong mode", "38U06");
            return;
        }

        /* depending on the phase decide what to do */
        switch(phase[0])
        {
            case Tbl.TBL_PRE_INIT:
                state_info = new local_ctx();
                ...
                break;
            case Tbl.TBL_INIT:
                /* Allocate scratchpad to retain data between iterations. */
                tbl.allocCtx(state_info);
                /* Preprocess data here. */
                ...
                break;
        }
    }
}

```

```

    case Tbl.TBL_BUILD:
        /* Get scratchpad and save data here. */
        state_info = (local_ctx)tbl.getCtxObject();
        ...
        /* Set SQLSTATE to "02000". */
        throw new SQLException("no more data", "02000");
        ...
        break;
    case Tbl.TBL_BUILD_EOF:
        /* Get scratchpad and build the result row here. */
        state_info = (local_ctx)tbl.getCtxObject();
        ...
        /* Or, if no more rows to build, set SQLSTATE to "02000". */
        throw new SQLException("no more data", "02000");
        ...
        break;
    case Tbl.TBL_FINI:
        /* Reset for the next set of data. */
        ...
        break;
    case Tbl.TBL_END:
        /* Everyone done. */
        ...
        break;
}
}
}

```

If you want to get a new row on each table UDF invocation, use `Tbl.getPhaseEx()` with the `Tbl.TBL_NEWROW` option. The following code excerpt shows an example of this usage.

```

import java.io.*;
import java.sql.*;
import com.teradata.fnc.*;

public class TableFunctions
{
    static boolean debug = true;

    public static void fnc_phase_new1(int in1, String in2, int[] out1,
String[] out2)
        throws SQLException, Exception
    {
        class GenCtx implements Serializable

```



```

{
    public int count;
    public GenCtx(){
    public GenCtx (int count)
    {
        this.count = count;
    }
}
try
{
    int [] phase = new int[1];
    GenCtx obj;
    Tbl tbl = new Tbl();

    // Only ask for the phase on each row
    int mode = tbl.getPhaseEx(phase, tbl.TBL_NOOPTIONS);

    if (mode != Tbl.TBL_MODE_VARY) {
        if (debug)
            System.err.println("Table function being called in unsupported context");
        throw new SQLException("Table function being called in unsupported
context", "U00006");
    }
    if (debug) System.err.println("MODE =" + mode);
    if (debug) System.err.println("Phase =" + phase[0]);

    switch(phase[0])
    {
        case Tbl.TBL_PRE_INIT:

            // Ask for a new row on each call to build
            tbl.getPhaseEx(phase, tbl.TBL_NEWROW);

            if (debug) System.err.println("Phase: TBL_PRE_INIT");
            //Init and allocate the context
            obj = new GenCtx();
            tbl.allocCtx(obj);
            tbl.setCtxObject(obj);

            trace ("\n In Pre Init");

            break;
        case Tbl.TBL_INIT:
            if (debug) System.err.println("Phase: TBL_INIT");

```

```

        //Get the context
        obj = (GenCtx)tbl.getCtxObject();
        //set value and store
        obj.count = 1;
        tbl.setCtxObject(obj);

        trace ("\n In Init");

        break;
    case Tbl.TBL_BUILD:
        if (debug) System.err.println("Phase: TBL_BUILD");
        //Get the context
        obj = (GenCtx)tbl.getCtxObject();
        if (debug){
            System.err.println("Phase: TBL_BUILD getObject =" +obj+", count="+obj.count);
        }
        trace ("\n In Build, input 1 is " + in1);
        out1[0] = in1;
        out2[0] = in2;
        if (debug)
            System.err.println("Phase: TBL_BUILD in1"+in1+", in2="+in2+",
out1[0]="+out1[0]+", out2[0]="+out2[0]);
        break;
    case Tbl.TBL_BUILD_EOF:
        trace ("\n In Build_EOF");
        if (debug) System.err.println("Phase: TBL_BUILD_EOF");
        break;
    case Tbl.TBL_FINI:
        trace ("\n In FINI");
        if (debug) System.err.println("Phase: TBL_FINISH");
        break;
    case Tbl.TBL_END:
        trace ("\n In END");
        if (debug) System.err.println("Phase: TBL_END");
        break;
    case Tbl.TBL_ABORT:
        trace ("\n In ABORT");
        if (debug) System.err.println("Phase: TBL_ABORT");
        break;
    default:
        throw new SQLException("Entering default phase.", "U0006");
    }
} catch (ClassNotFoundException e)

```

```

    {
        e.printStackTrace();
    } catch(IOException e)
    {
        e.printStackTrace();
    }
}
}

```

## Table UDFs that Retain Data Between Iterations

Table UDFs can use intermediate storage to retain data between iterations.

In the TBL\_PRE\_INIT phase, a table function usually allocates intermediate storage space by calling `com.teradata.fnc.Tbl.allocCtx()`, passing in an initialized instance of the class that the table function uses to hold the data it wants to retain. In later phases, the table function calls `com.teradata.fnc.Tbl.setCtxObject()` to write data to the previously allocated intermediate storage space and `com.teradata.fnc.Tbl.getCtxObject()` to retrieve the data and continue the processing.

## Classes that Hold the Data to Retain Between Iterations

To enable `Tbl.allocCtx()` to serialize a class object into a binary stream and allocate storage for it, the class that the Java table UDF uses to hold the intermediate data must implement the `java.io.Serializable` interface. (The class must be serializable.) The size of the serialized binary stream determines the size of the allocated intermediate storage and it should be big enough to hold all the data of the object.

In Java, a serializable class overwrites the default methods `writeObject(java.io.ObjectOutputStream out)` and `readObject(java.io.ObjectInputStream in)` with its own customized implementation. An object of such a class automatically calls `writeObject()` during its serialization and `readObject()` during its deserialization.

If the object that the Java table UDF uses to hold intermediate data has object type fields, the UDF must initialize those fields to non-null values before calling `Tbl.allocCtx()`. One way to do this is to implement the `writeObject()` method to check the fields for nulls and initialize them if needed.

Similarly, if the object has array type fields, the UDF must initialize the array itself and each element of the array to non-null values before calling `Tbl.allocCtx()`.

## Using a Byte Array to Hold Data Between Iterations

For better performance, you can use a byte array to hold the data to retain between iterations in Java table UDFs.

The methods for allocating the context object, setting it and retrieving it are:

- `Tbl.allocCtx(int)` without a return value.
- `Tbl.setCtxObject(byte[])` without a return value.
- `getCtxObject(byte[])` with a return value of `byte[]`.

Note that you will have to provide methods for serializing and deserializing the context object accordingly.

The following is an example that uses a byte array to hold an integer value as the context object. The supporting methods `byteArrayToInt(byte [])` and `intToByteArray(int)` are provided as needed.

```
import com.teradata.fnc.Tbl;
import java.sql.*;

public class Measure
{
    public static final int byteArrayToInt(byte [] b)
    {
        return (b[0] << 24)
            + ((b[1] & 0xFF) << 16)
            + ((b[2] & 0xFF) << 8)
            + (b[3] & 0xFF);
    }
    public static final byte[] intToByteArray(int value)
    {
        return new byte[] {
            (byte)(value >>> 24),
            (byte)(value >>> 16),
            (byte)(value >>> 8),
            (byte)value };
    }

    public static void j_noop_table1_bytearray(double a, double[] c) throws SQLException
    {
        byte[] ctxByteArr = {0, 0, 0, 0}; // scratch pad
        int count;
        try
        {
            int [] phase = new int[1];
            Tbl tbl = new Tbl();
            int mode = tbl.getPhase(phase);
            if (mode!=Tbl.TBL_MODE_VARY) throw new SQLException("Wrong Mode", "U0001");
            switch(phase[0])
            {
                case Tbl.TBL_PRE_INIT:
                    tbl.allocCtx(4); // 4-byte integer
                    break;

                case Tbl.TBL_INIT:
                    count = 1;
                    ctxByteArr = intToByteArray(count);
                    tbl.setCtxObject(ctxByteArr);
                    break;

                case Tbl.TBL_BUILD:
                    ctxByteArr = (byte[])tbl.getCtxObject(ctxByteArr);
                    count = byteArrayToInt(ctxByteArr);
                    if (count == 0) throw new SQLException("no more data", "02000");

                    c[0] = a;

                    count--;
                    ctxByteArr = intToByteArray(count);
                    tbl.setCtxObject(ctxByteArr);
                    break;

                case Tbl.TBL_FINI:
                case Tbl.TBL_END:
                case Tbl.TBL_ABORT:
            }
        }
    }
}
```

```

        default:
            break;
    }
} catch (ClassNotFoundException e) {
    throw new SQLException("ClassNotFoundException", "U0002");
} catch (IOException e) {
    throw new SQLException("IOException", "U0003");
}
}
}

```

## Clob and Blob Classes

Teradata implements `java.sql.Clob` and `java.sql.Blob` as serializable classes. Thus, a table function can use them, passing their objects between iterations of table function execution.

However, writing a table UDF involves special considerations when the table UDF uses Clob or Blob objects to hold the data in the intermediate storage.

Consider a table UDF that retains data between iterations using an `item_clob` object:

```

class item_clob implements Serializable
{
    int id;
    java.sql.Clob myclob;

    ...
}

```

To ensure that `Tbl.allocCtx()` allocates intermediate storage space big enough to hold all the data of the object for all iterations of the table function, the table UDF can implement the `writeObject()` method to write a dummy byte array into the serialized binary stream if member `myclob` is null.

The size of the dummy byte array is 64, which is the size of a fully serialized Teradata `java.sql.Clob` and `java.sql.Blob` object.

```

class item_clob implements Serializable
{
    int id;
    java.sql.Clob myclob;

    private void writeObject(java.io.ObjectOutputStream out)
    throws IOException
    {
        byte[] dummy = new byte[64];
        out.defaultWriteObject();
        /* Allocate storage for lob by writing the dummy bytes */
        /* into its serialized binary stream if myclob is null. */
        if (myclob == null)

```

```

        out.write(dummy);
    }

    private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        in.defaultReadObject();
    }
}

```

For a complete example, see [Example: Table UDF Using Clob to Hold Data in Intermediate Storage](#).

## Example: Table UDF Using Clob to Hold Data in Intermediate Storage

For this example, consider the table `clob_tbl`, where the definition and data is as follows:

```

CREATE TABLE clob_tbl(id INTEGER,
    clob_column_2 CLOB(20)) PRIMARY INDEX(id);

INSERT clob_tbl(1, 'First 3 New Rows');
INSERT clob_tbl(2, 'Second 3 New Rows');

```

The table function `ClobTbl` is created as follows:

```

CALL SQLJ.INSTALL_JAR('CJ!ClobFunctions.jar', 'ClobFunctions',0);

REPLACE FUNCTION ClobTbl(in_parm CLOB(20))
    RETURNS TABLE (id    INTEGER,

                    clobcontent  VARCHAR(20))
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'ClobFunctions:ClobFunctions.ClobTbl';

```

The source code of the function `ClobTbl()` appears below. The following comments highlight code of interest.

Comments	Details
<pre> /** PROBLEM **/ ... </pre>	<p>These comments delimit code that instantiates a <code>local_ctx_clob</code> class object <code>clob_info</code> and then calls <code>allocCtx(clob_info)</code> to allocate the storage space for it. The method <code>allocCtx()</code> serializes <code>clob_info</code> into a binary stream and reserves the storage with the size equal to the size of the binary stream. The class <code>local_</code></p>

Comments	Details
<pre>/** END PROBLEM **/</pre>	<p>ctx_clob has an array of java.sql.Clob[] type. But in the TBL_PRE_INIT phase, all entries in the java.sql.Clob[] array default to null. Thus, when the call to allocCtx() serializes clob_info, there is no java.sql.Clob object for serialization. This results in a much smaller serialized binary stream than what is needed. Thus, the storage space allocated is also smaller than what is needed. Later, when the first lob data coming is processed in the TBL_INIT phase, the call Tbl.setCtxObject() causes an error because the UDF is trying to write more data than what is available in the reserved storage space.</p>
<pre>/** SOLUTION **/ ... /** END SOLUTION **/</pre>	<p>These comments delimit code that solves the problem. The code checks each item_clob object and if its member myclob is null, then it writes a dummy byte array into the serialized binary stream so that its size can be big enough for later usage. The size of the dummy byte array is 64, which is the size of a fully serialized Teradata java.sql.Clob and java.sql.Blob object.</p>

```
import java.io.*;
import java.sql.*;
import com.teradata.fnc.*;

class item_clob implements Serializable
{
    int id;
    java.sql.Clob myclob;

    private void writeObject(java.io.ObjectOutputStream out)
    throws IOException
    {
        byte[] dummy = new byte[64];
        out.defaultWriteObject();
        /***** SOLUTION *****/
        /* Allocate storage for lob by writing the dummy bytes */
        /* into its serialized binary stream if myclob is null. */
        if (myclob == null)
        {
            out.write(dummy);
        }
        /***** END SOLUTION *****/
    }
    private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException
    {
        in.defaultReadObject();
    }
}

class local_ctx_clob implements Serializable
```

```

{
    int Num_Row;
    int Cur_Row;
    item_clob[] Clob_List;
    public local_ctx_clob(){}
    public local_ctx_clob(int NR, int CR, item_clob[] Clob_List)
    {
        this.Num_Row=NR;
        this.Cur_Row=CR;
        this.Clob_List=Clob_List;
    }
}

public class ClobFunctions
{
    static int cnt = 0;

    /* ClobTbl uses each lob value in clob_tbl.clob_column_2 */
    /* to generate 3 new rows for the return table.          */
    public static void ClobTbl( java.sql.Clob in_parm, int[] id,
                               String[] clobcontent )

    throws SQLException
    {
        local_ctx_clob clob_info;
        int[]          phase = new int[1];
        Tbl            tbl = new Tbl();

        try
        {
            switch (tbl.getPhase(phase))
            {
                case Tbl.TBL_MODE_CONST:
                    throw new SQLException("not for const mode", "U0006");
                case Tbl.TBL_MODE_VARY:
                    switch(phase[0])
                    {
                        case Tbl.TBL_PRE_INIT:
                            {
                                /****** PROBLEM *****/
                                /* Allocate storage space to hold intermediate */
                                /* data which is used to generate 3 new rows.  */
                                item_clob[] clob_arr = new item_clob[3];
                                for( int i=0; i<clob_arr.length; i++)
                                    clob_arr[i]=new item_clob();
                            }
                        }
                    }
            }
        }
    }
}

```



```

        clob_info = new local_ctx_clob(3, 0, clob_arr);
        tbl.allocCtx(clob_info);
        /******* END PROBLEM *****/
    }
    break;

case Tbl.TBL_INIT:
{
    /* Each lob passed in is stored in an      */
    /* intermediate structure that is used to   */
    /* construct 3 new rows in TBL_BUILD phase. */
    item_clob[] clob_arr = new item_clob[3];
    for( int i=0; i<clob_arr.length; i++) {
        clob_arr[i] = new item_clob();
        clob_arr[i].myclob = in_parm;
        if (in_parm != null) clob_arr[i].id = cnt++;
    }
    clob_info = new local_ctx_clob(3, 0, clob_arr);
    /* Store the intermediate data */
    tbl.setCtxObject(clob_info);
}
break;

case Tbl.TBL_BUILD:
{
    /* Get intermediate data stored in TBL_INIT */
    /* phase and use it to generate 3 rows */
    clob_info = (local_ctx_clob)tbl.getCtxObject();
    if ( clob_info.Cur_Row >= clob_info.Num_Row ||
        clob_info.Clob_List[0].myclob == null) {
        throw new SQLException("no more data","02000");
    } else {
        id[0] =
            clob_info.Clob_List[clob_info.Cur_Row].id;
        Reader R =
            clob_info.Clob_List[clob_info.Cur_Row].myclob.getChara
cterStream();

        char[] chars = new char[20];
        R.read(chars);
        R.close();
        clobcontent[0] = (new String(chars)).trim();
        clob_info.Cur_Row++;
    }
    tbl.setCtxObject(clob_info);
}

```

```

    }
    break;

    case Tbl.TBL_FINI:
        clob_info = (local_ctx_clob)tbl.getCtxObject();
        clob_info.Cur_Row = 0;
        clob_info.Num_Row = 0;
        clob_info.Clob_List= null;
        break;

    case Tbl.TBL_END:
        clob_info = (local_ctx_clob)tbl.getCtxObject();
        break;

    case Tbl.TBL_ABORT:
        clob_info = (local_ctx_clob)tbl.getCtxObject();
        break;

    }
    return;
}

}catch(ClassNotFoundException e){
    throw new SQLException("Class not located Error", "U0006");
}catch(StreamCorruptedException e){
    throw new SQLException("Stream Corrupt Error", "U0006");
}catch(IOException e){
    e.printStackTrace();
}

}
}
}

```

The content of table `clob_tbl` is:

```
BTEQ -- Enter your DBC/SQL request or BTEQ command:

sel * from clob_tbl;

*** Query completed. 2 rows found. 2 columns returned.
*** Total elapsed time was 1 second.

      id  clob_column_2
-----
```

```

1 First 3 New Rows
2 Second 3 New Rows

```

Without the solution code, calling ClobTbl() produces an error.

```

BTEQ -- Enter your DBC/SQL request or BTEQ command:

SELECT new_tbl.id, new_tbl.clobcontent
      FROM TABLE (ClobTbl(clob_tbl.clob_column_2)) AS new_tbl
      ORDER by new_tbl.id;

*** Failure 7828 Unexpected Java Exception SQLSTATE 38000: An
java.lang.Error (Tbl.setCtxObject failed because object was larger
than general scratchpad.) exception was thrown.
          Statement# 1, Info =0
*** Total elapsed time was 3 seconds.

```

With the solution code, calling ClobTbl() produces the correct answer:

```

BTEQ -- Enter your DBC/SQL request or BTEQ command:

SELECT new_tbl.id, new_tbl.clobcontent
      FROM TABLE (ClobTbl(clob_tbl.clob_column_2)) AS new_tbl
      ORDER by new_tbl.id;

*** Query completed. 6 rows found. 2 columns returned.
*** Total elapsed time was 1 second.

```

id	clobcontent
0	First 3 New Rows
1	First 3 New Rows
2	First 3 New Rows
3	Second 3 New Rows
4	Second 3 New Rows
5	Second 3 New Rows

## Table Operators

The guidelines for implementing Java table operators are similar to the guidelines for implementing C/C++ table operators. For more information, see [Table Operators](#).

From the Java perspective, the input and output streams come preopened and you do not have to close the input or output stream because the end of the function is an implicit close. You can close a stream only

once. Also, you do not have to read the entire input stream. If you get End Of Data on the input stream, an exception is thrown.

---

**Note:**

Calling `.first()` on a `TeradataResultSet()` rewinds the cursor before the first row instead of to the first row as specified in the Java documentation for `java.sql.ResultSet`. Therefore, `TeradataResultSet.first()` is more like `java.sql.ResultSet.beforeFirst()`.

---

Teradata provides the following Java application classes to support table operators:

- `ArrayTypeInfo`
- `ColumnDefinition`
- `InputInfo`
- An extension of `ResultSet`
- `RuntimeContract`
- `StreamFormat`
- `UDTBaseInfo`
- `AMPIInfo`
- `NodeInfo`

The core class for Java SQLTABLE parameter style is `RuntimeContract`. You define a class with a constructor and an iterator method that is used during the row processing. The constructor acts as the contract function.

Java table operators run in the protected mode Java server just like regular Java UDFs. This environment is multithreaded so you must write the user code to be thread safe.

The rows provided to the table operator are buffered for input and output to improve performance.

## Java Table Operator Metadata Mapping

Data type metadata, including UDT and CDT metadata, can be passed to a table operator contract function.

Note the difference between the terms *mapping* and *transformed* type as follows:

### Mapping Type

The External Type code that is used in the contract function to identify the data type of a column.

### Transform Type

The predefined data type that a UDT or CDT value is converted to by invoking the transform function.

By default, all of the UDT or CDT data values are passed in their default transform form to the operator. This may correspond to either the SQL transform type of the UDT or another predefined type. The behavior is

explicitly determined for each individual UDT or CDT type. For Java table operators, an option is provided to pass some UDTs and CDTs in an untransformed or atomized form.

Each of the UDT or CDT type is also mapped to an External Type code in the table operator contract function. Additional information about each type is passed in the `UDT_BaseInfo_t` structure for C/C++ table operators or the `com.teradata.fnc.runtime.UDTBaseInfo` class for Java table operators.

The following table shows the metadata mapping in the contract function from Teradata SQL type to the External Type code for UDTs and CDTs.

SQL UDT or Complex Data Type	External Type Code
ARRAY/VARRAY	ARRAY_DT
DATASET in the Avro storage format	DATASET_AVRO_DT
DATASET in the CSV storage format	DATASET_CSV_DT
Geospatial – MBB	MBB_DT
Geospatial – MBR	MBR_DT
Geospatial – ST_Geometry	ST_GEOMETRY_DT
JSON	JSON_DT
Period type	PERIOD_DT
UDT (Distinct)	UDT_DT
UDT (Structured)	UDT_DT
XML	XML_DT

## Passing UDT and CDT Columns To and From a Table Operator

The contract function for a table operator describes the input and output columns to be passed and retrieved. The following sections provide detail on how UDT and CDT columns may be passed to and from a table operator.

## Retrieving UDT and CDT Input Metadata

The UDT and CDT metadata can be retrieved in the contract function in two ways:

- You can call the `RuntimeContract.getBaseInfo()` method. This method returns an array of `UDTBaseInfo` objects for the `ColumnDefinition` objects passed in. This method can only be invoked in the contract function.
- You can use the `InputInfo` class `getUDTMetaData()` method to retrieve the UDT and CDT metadata in both the contract and execute functions. The method returns an array of `UDTBaseInfo` objects for the input columns.

## Setting UDT and CDT Output Metadata

Each output column is described using an instance of the class `ColumnDefinition`.

### Related Information:

[Java Table Operator Default Transform Behavior and Transform Off Format](#)

## Example: Java Table Operator With UDT Input and Output Columns

This is an example of a Java table operator that can accept input columns and return output columns of UDT or complex types. The contract function shows how each of the UDT or CDT type can be handled.

```
/*
CREATE FUNCTION jtblop_udtvals()
RETURNS TABLE VARYING USING FUNCTION jtblop_udtvals_contract
LANGUAGE JAVA NO SQL
PARAMETER STYLE SQLTABLE
EXTERNAL NAME 'TBLOPUDT_JAR:jtblop_udtvals.execute';
*/

public class jtblop_udtvals implements TableOperator
{
    public jtblop_udtvals() {};

    public int contract(RuntimeContract contract, ResultSet rsin[], ResultSet
rsout[])
    throws SQLException
    {
        /* Get number of input streams */
        int incount = contract.getInputInfo().getIncount();

        /* Get input columns. */
        Metadata iCols;
        int total_colcount = 0;
        for(int i = 0; i < incount; i++)
        {
            if (rsin[i] != null)
            {
                iCols = ((TeradataResultSet)rsin[i]).getTeradataMetaData();
                total_colcount += iCols.getColumnCount();
            }
        }
    }
}
```

```

/* Create an equal number of output columns. */

ColumnDefinition OutCols[] = new ColumnDefinition[total_colcount];

/* Copy input columns to output columns. */
int ocount = 0;
for(int j = 0; j < incount; j++)
{
    Metadata inCols = ((TeradataResultSet)rsin[j]).getTeradataMetaMeta();
    UDTBaseInfo udtbaseinfo[] = contract.getInputInfo().getUDTMetadata(j);
    for (int i=0;i<inCols.getColumnCount();i++)
    {
        OutCols[ocount]=new ColumnDefinition("O"+ocount,
inCols.getTeradataColumnType(i+1));

OutCols[ocount].setDisplayLength(inCols.getColumnDisplaySize(i+1));

        TeradataType t = TeradataType.get(inCols.getTeradataColumnType(i+1));
        switch (t)
        {
            case VARCHAR_DT:
            case CHAR_DT:
                OutCols[ocount].setCharset(inCols.getPrecision(i+1));
                break;
            case BLOB_REFERENCE_DT:
            case CLOB_REFERENCE_DT:
            case VARBYTE_DT:
            case XML_DT:
            case JSON_DT:
            case ST_GEOMETRY_DT:
                OutCols[ocount].setCharset(inCols.getPrecision(i+1));

OutCols[ocount].setDisplayLength(inCols.getColumnDisplaySize(i+1));
                break;
            case DECIMAL1_DT:
            case DECIMAL2_DT:
            case DECIMAL4_DT:
            case DECIMAL8_DT:
            case DECIMAL16_DT:
                OutCols[ocount].setPrecision(inCols.getPrecision(i+1));
                OutCols[ocount].setScale(inCols.getScale(i+1));
            case NUMBER_DT:
                OutCols[ocount].setPrecision(inCols.getPrecision(i+1));

```

```

        OutCols[ocount].setScale(inCols.getScale(i+1));
        break;
    case TIME_DT:
    case TIMESTAMP_DT:
    case TIME_WTZ_DT:
    case TIMESTAMP_WTZ_DT:
    case PERIOD_DT:
        OutCols[ocount].setPrecision(inCols.getPrecision(i+1));
        break;
    case INTERVAL_SECOND_DT:
    case INTERVAL_DTS_DT:
    case INTERVAL HTS_DT:
    case INTERVAL_MTS_DT:
    case INTERVAL_DTM_DT:    // DAY TO MINUTE
        OutCols[ocount].setPrecision(inCols.getPrecision(i+1));
        OutCols[ocount].setScale(inCols.getScale(i+1));
        break;
    case ARRAY_DT:
    case UDT_DT:
    case MBR_DT:
    case MBB_DT:
        OutCols[ocount].setUdtName(inCols.getUdtTypeName(i+1));

    }
    OutCols[ocount].setPeriodType(inCols.getPeriodType(i+1));
    ocount++;
}
}

/* Define output columns. */
contract.setOutputInfo(0, OutCols);
contract.complete();
return 1;
}

public void execute(RuntimeContract contract, ResultSet rsin[], ResultSet
rsout[])
throws SQLException
{
    /* Tblp Execute code */
} /* execute */
}

```



## Java Table Operator Default Transform Behavior and Transform Off Format

### Default Transform Behavior for UDT and CDT Data Values

The input UDT and CDT data values sent to the table operator are in their default transform form. This may correspond to either the SQL transform type of the UDT or another predefined type. The UDT and CDT values are converted to their corresponding transform type by invoking the default transform function defined for the UDT or CDT. The following table shows the transform types for UDT and CDT data types passed to a table operator.

This mode is well suited for a connector-type of operator (T2T or T2M) that does import and export of data values.

Note that some UDTs and CDTs (such as XML, ST\_Geometry, DATASET, and JSON) support multiple transform groups. However, for table operators, the CDT values will always be sent using the *default* transform type. For instance, a JSON data value will always be sent as a CLOB.

SQL UDT or Complex Data Type	SQL Data Type Mapping	Java Data Type
ARRAY/ VARRAY	VARCHAR (the defined transform type)	java.lang.String
BSON	BLOB or CLOB	<ul style="list-style-type: none"> <li>• java.sql.Blob</li> <li>• java.sql.Clob</li> </ul>
DATASET – Avro	BLOB	java.sql.Blob
DATASET – CSV	CLOB	java.sql.Clob
Geospatial – MBB	VARCHAR	java.lang.String
Geospatial – MBR	VARCHAR	java.lang.String
Geospatial – ST_Geometry	CLOB	java.sql.Clob
JSON	CLOB	java.sql.Clob
Period	VARCHAR	java.lang.String
UDT (Distinct)	Predefined data type that the UDT is based on	Primitive Java data type For example: <ul style="list-style-type: none"> <li>• int</li> <li>• short</li> <li>• java.lang.String</li> <li>• java.sql.Clob</li> <li>• java.sql.Blob</li> </ul>
UDT (Structured)	Predefined data type as defined in CREATE TRANSFORM	Primitive Java data type

SQL UDT or Complex Data Type	SQL Data Type Mapping	Java Data Type
		For example: <ul style="list-style-type: none"> <li>• int</li> <li>• short</li> <li>• java.lang.String</li> <li>• java.sql.Clob</li> <li>• java.sql.Blob</li> </ul>
XML	CLOB	java.sql.Clob

### Setting the Transforms Off Option

If a table operator needs to work on table data and be able to access and retrieve individual elements of a complex type, then the transform type is not suitable. Therefore, an option is provided that allows a table operator to request Period and Array values to be sent in a *transforms off*, or atomized, form.

The RuntimeContract class setFormat() method can be used to set the format attributes corresponding to PDTTTransformsOff and ArrayTransformsOff. These format attributes must be set to TRUE if Period and Array values will be sent and received in the untransformed form.

---

#### Note:

You cannot enable the Transforms Off option for structured UDTs.

---

### Transforms Off Data Values

When the Transforms Off mode is TRUE, Period values are sent as java.sql.Struct objects and Arrays are sent as java.sql.Array objects in the Input ResultSet sent to the operator. The operator can then use methods of the java.sql.Struct and java.sql.Array classes to manipulate the objects.

Similarly, the operator can construct java.sql.Struct and java.sql.Array objects for the output data values.

## Getting and Setting the Time Zone in a Java Table Operator

You can get or set the time zone value in a Java table operator.

To get the time zone value, do the following:

1. Call getTime() or getTimestamp() and check that the return type is TeradataTime or TeradataTimestamp.
2. If the return type is TeradataTime or TeradataTimestamp, call getTimeZone() on the object to get the time zone.

To set the time zone value, do the following:

1. Create a TeradataTime or TeradataTimestamp object.
2. Call setTimeZone() on the TeradataTime or TeradataTimestamp object, and pass the result to updateTime () or updateTimestamp() as the second parameter.

**Note:**

TeradataTime and TeradataTimestamp are used to get and set the time zone in Time with Time Zone and Timestamp with Time Zone values in the input or output rows. They are used only with Java table operators. For information about using Time with Time Zone and Timestamp with Time Zone as parameter and return types in a Java table operator, see [SQL to Java Data Type Mapping](#).

## Example: Getting and Setting the Time Zone in a Java Table Operator

The following code excerpt shows the usage of the Java TeradataTime and TeradataTimestamp data types and the getTimeZone and setTimeZone methods for getting and setting the time zone value in a Java table operator.

```
String timezone = "America/New_York";
Calendar c = Calendar.getInstance();
TimeZone tz = TimeZone.getTimeZone(timezone);
c.setTimeZone(tz);

TeradataTimestamp tts = new TeradataTimestamp( c.getTimeInMillis());
tts.setTimeZone(tz);

TeradataTime tTime = new TeradataTime(c.getTimeInMillis());
tTime.setTimeZone(tz);

if(outObjs[i] instanceof TeradataTimestamp){
    TeradataTimestamp outtts = (TeradataTimestamp)outObjs[i];
    java.util.TimeZone tz = outtts.getTimeZone();
}
else if(outObjs[i] instanceof TeradataTime){
    TeradataTime outtTime = (TeradataTime)outObjs[i];
    java.util.TimeZone tz = outtTime.getTimeZone();
}
else if(outObjs[i] instanceof Time){
    Time outtTime = (Time)outObjs[i];
    printSimpleTimeReading(outtTime);
}
else if(outObjs[i] instanceof Timestamp){
    Timestamp outtTime = (Timestamp)outObjs[i];
    printSimpleTimestampReading(outtTime);
}
```

## Related Information

FOR more information on ...	SEE ...
the SQL definition for a table operator	"CREATE FUNCTION (Table Form)" in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.
invoking a table operator in an SQL query	the FROM clause of the SELECT statement in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the Java application classes available to table operator and contract function writers	<a href="#">Java Application Classes</a> .
the TeradataTime and TeradataTimestamp Java classes	<ul style="list-style-type: none"> <li>• <a href="#">com.teradata.fnc.value.TeradataTime</a></li> <li>• <a href="#">com.teradata.fnc.value.TeradataTimestamp</a></li> </ul>
example Java code for a table operator	<a href="#">Java Table Operator</a> .

## Exception Handling

The Java method that you write to implement an external routine (UDF or external stored procedure) can use the `SQLException` class to throw an exception. The method may also need to handle exceptions that are thrown when the method uses a Teradata application class and an error occurs.

## Returning SQLSTATE Values

A Java UDF can throw a `java.lang.SQLException` with an exception number and message that are returned and used to set the SQLSTATE value and error message.

Valid values for the `SQLState` field of the `SQLException` are in the range 38U00 to 38U99.

Here is an example:

```
public class UDFExample {

    public static Integer fact( Integer x )
    throws SQLException
    {
        int factresult = 0;
        if (x != null)
            factresult = 1;
        else
            throw new SQLException("Input value not valid", "38U01");
        ...
        return new Integer(factresult);
    }
}
```

```

    }

    ...

}

```

SQLException is the only valid exception that Java external routines can throw.

## Checking Exceptions Returned From Teradata Application Classes

Teradata provides several application classes that external routines may need to use. A method defined for a Teradata application class can return Teradata specific errors by throwing an SQLException where the value of the SQLState field is TS000.

Please note that in the case where an SQLState of TS000, a Teradata Specific Condition, there is an associated Teradata specific error code that may be accessed via the `getError()` method of the SQLException class. As an implementation detail only, these error codes are allocated like normal Teradata error codes and error text.

Here is an example of using the Tbl class and handling an exception:

```

public static void ExceptionCatch() {
    try {
        int len = Tbl.getScale(); /* Throws an SQLException. */
    } catch (SQLException e) {
        if (e.getSQLState().equals("TS000")) {
            /* Check Teradata exception.*/
            int TeradataCode = e.getErrorCode(); /* Get specific error. */
        }
    }
}

```

If an external routine does not handle a TS000 SQLException, an SQLSTATE value of 39001 is returned. Here is an example where the `getScale()` method of the Tbl class returned an exception to a table UDF that did not handle it:

```

*** Failure 7827 Java SQL Exception SQLSTATE 39001: Invalid SQL state (TS000
[Error 7850]: Tbl.getScale(int) is invalid for the type of column intended).

```

For details on the error codes that Teradata application classes can return, see [Java Application Classes](#).

## Handling Exceptions From Java External Routines

The following rules apply when Vantage catches an exception from a Java external routine:

IF the class of the exception is...	THEN...
not SQLException	<p>Vantage returns an SQLSTATE value of 38000. Here is an example:</p> <pre>*** Failure: Statement = 1  Info 0       Code 7828: Unexpected Java Exception SQLSTATE 38000:       An java.lang.NullPointerException exception was thrown.</pre> <p>This typically occurs when the external routine fails to catch an exception.</p>
SQLException	<p>If the SQLState field is five digits, where the first two digits are 38, then Vantage returns the SQLSTATE value. Here is an example:</p> <pre>SEL o_id, std_dev(o_amount); FROM orders;</pre> <pre>*** Failure 7825 in UDF/XSP/UDM UDFExample.std_dev: SQLSTATE 38U01:  User Exception Text.</pre> <p>This is considered a properly returned error from a UDF.</p> <p>If the SQLState field is out of range, then Vantage returns an SQLSTATE value of 39001. Here is an example:</p> <pre>SEL o_id, std_dev(o_amount); FROM orders;</pre> <pre>*** Failure 7827 Java SQL Exception SQLSTATE 39001: Invalid SQL state (45100).</pre>

## Compiling the Source Code

To compile the source code for your Java routine, you need javFnc.jar, the Teradata Java external stored procedure and UDF runtime library.

IF you compile the source code on ...	THEN ...
your client system	download the runtime library from <a href="#">Teradata Downloads</a> .
the database system	the runtime library is already available.

Add the path to the runtime library to your class path (or use the -classpath option of the Java compiler). For details on the location of the runtime library on a database system, see [Class Path](#).

For example, suppose you downloaded javFnc.jar to C:\java\_udf on your Windows client. You can compile the source code in UDFExample.java as follows:

```
javac -classpath C:\java_udf\javFnc.jar UDFExample.java
```

The version of the class file that you generate when you compile the Java source code must be compatible with the JRE on the database system.

For example, suppose the database uses JRE 1.8 and your client system uses JDK 8.0. To compile the source code in UDFExample.java on your client system, you can use the -target option of the Java compiler.

```
javac -target 1.8 -classpath C:\java_udf\javFnc.jar UDFExample.java
```

To determine the version of the JRE on the database system, use the `cufconfig` utility to see where the JRE was installed. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Registering the JAR or ZIP File

After you write and test the Java source code for a UDF, you place the resulting class or classes in an archive (JAR or ZIP) file.

Next, you call the `SQLJ.INSTALL_JAR` external stored procedure to register the archive file and its classes with the database, specifying an SQL identifier for the archive file.

If your source code is on a client system, the SQL identifier that you specify is the same name that you use later in the `EXTERNAL NAME` clause of the `CREATE FUNCTION` or `REPLACE FUNCTION` statement to define the SQL function.

For example, consider a JAR file called `reports.jar` on a Windows client in the directory `C:\java_udf`.

The following statements register the JAR file with the JUDF database and create an SQL identifier called `Report_JAR` for the JAR file:

```
DATABASE JUDF;
CALL SQLJ.INSTALL_JAR('C:\java_udf\reports.jar', 'Report_JAR', 0);
```

For more information on how to register archive files for a Java external routine, see [Registering and Distributing JAR and ZIP Files for Java External Routines](#).

## Defining the SQL Function

After you register the JAR or ZIP file containing the class or classes that implement the UDF, you can define the SQL function with the `CREATE FUNCTION` or `REPLACE FUNCTION` statement.

### Database for the Function

When you execute the `CREATE FUNCTION` or `REPLACE FUNCTION` statement to define a function that is implemented by a class in the JAR or ZIP file, you must define it in the same database in which the JAR or ZIP file was registered.

## Specifying the Class Name, Method Name, and JAR or ZIP Identifier

The CREATE FUNCTION and REPLACE FUNCTION statements provide the EXTERNAL NAME clause for specifying the identifier of the registered JAR or ZIP file, and the class and method name in the JAR or ZIP file that implements the function.

Consider the following CREATE FUNCTION statement:

```
CREATE FUNCTION factorial (x INTEGER)
  RETURNS INTEGER
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'JarUDF:UDFExample.fact';
```

where:

THIS part of the EXTERNAL NAME clause ...	Specifies the ...
JarUDF	identifier of the registered JAR file that was provided to the SQLJ.INSTALL_JAR procedure.
UDFExample	name of the class in the registered JAR file that implements the Java UDF.
fact	method name in the specified class that Vantage invokes when the UDF is specified in an SQL statement.

## Related Information

FOR more information on the ...	SEE ...
CREATE FUNCTION and REPLACE FUNCTION statements	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
privileges that apply to UDFs	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Debugging a User-Defined Function

Debugging a Java method that Vantage executes as a UDF is limited.



## Debugging Outside the Database

The best practice is to develop and test the method outside the database. You can use your own debugging tools to verify functionality.

Be sure your tests include the following:

- Limit checks on the input values
- Limit checks on the return value (for scalar and aggregate UDFs) or on the output argument values (for table UDFs)
- Proper handling of NULLs
- Exception handling

## Using Trace Tables

If debugging outside the database is not sufficient, you can use trace tables to get trace diagnostic output. Debugging UDFs using trace tables is similar to debugging external stored procedures using trace tables. For details, see [Debugging Using Trace Tables](#).

## Resolving UDF Server Setup Errors

When an external routine like an UDF, UDM, or external stored procedure is called, a UDF server process is acquired from the UDF server pool to execute the external routine. If there are no UDF server processes in the pool or if all of the processes in the pool are busy, then the system tries to start a new UDF server process for the request.

The startup of the new UDF server usually takes some time, especially if the UDF server is for executing Java external routines, or if the system is very busy. If the new UDF server cannot be started within the default time limit, the query that contains the UDF, UDM, or external procedure call is aborted, and you may receive a 7583 error indicating that the UDF server setup encountered a problem. The system log may also record a 7820 error specifying that the UDF server could not stay up long enough for initialization.

If you are experiencing these errors, you can contact Teradata Support Center personnel to adjust the time limit allowed for starting a new UDF server process. For details, see the information about the Cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## UDF Invocation

A scalar UDF can appear anywhere a standard SQL scalar function can appear and an aggregate UDF can appear almost anywhere a standard SQL aggregate function can appear.

A table function can only appear in the FROM clause of an SQL SELECT statement. The SELECT statement containing the table function can appear as a subquery.

## Restrictions

Aggregate UDFs cannot appear in recursive queries.

## Required Privileges

To invoke a UDF, you must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

## Argument List

The arguments in the function call must appear as comma-separated expressions in the same order as the function declaration parameters.

The arguments in the function call must be compatible with the parameter declarations in the function definition of an existing function, and must fit into the compatible type without a possible loss of information. For example, a BYTEINT argument in a function call is compatible with an INTEGER parameter declaration in the function definition, and also fits into the INTEGER type without any loss of information.

To pass an argument that is not compatible with the corresponding parameter type, explicitly convert the argument to the proper type in the function call.

## Behavior When Using NULL as a Literal Argument

To specify whether Vantage invokes a Java method when an SQL statement passes NULL as a UDF argument, the CREATE FUNCTION or REPLACE FUNCTION statement can include the RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT clause.

IF an input argument is the NULL keyword and the corresponding CREATE FUNCTION statement ...	THEN ...
specifies RETURNS NULL ON NULL INPUT	the method is not evaluated and the result is always NULL.
specifies CALLED ON NULL INPUT	the method is passed a null input argument.
does not specify either clause	

To properly handle the NULL literal as an input argument, a Java UDF cannot use the default mapping convention if the SQL data types in the parameter list of the CREATE FUNCTION or REPLACE FUNCTION statement map to Java primitives.

To override the default mapping and map SQL data types to Java classes that can handle the NULL literal as an input argument, the EXTERNAL NAME clause in the CREATE FUNCTION or REPLACE FUNCTION statement must explicitly specify the mapping in the parameter list of the Java method.

For details on how SQL data types map to Java data types, see [SQL Data Type Mapping](#). For an example that shows how to override the default mapping, see [Example: Overriding Default Parameter Mapping to Handle NULLs](#).

## Invoking UDFs with TD\_ANYTYPE Result Parameters

When invoking a scalar or aggregate UDF that is defined with a TD\_ANYTYPE result parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clauses to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column. You must enclose the UDF invocation in parenthesis if you use the RETURNS or RETURNS STYLE clauses.

For detailed information, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).

## Using Non-Deterministic UDFs as Conditions on an Index

A non-deterministic UDF is a UDF that does not always return identical results for identical inputs. If you omit the DETERMINISTIC clause in the CREATE FUNCTION or REPLACE FUNCTION statement, or if you specify the NOT DETERMINISTIC clause, the UDF is considered to be non-deterministic. Because non-deterministic UDFs are evaluated for each selected row, a condition on an index column that includes a non-deterministic UDF results in an all-AMP operation.

For example, consider the following table definition:

```
CREATE TABLE t1
  (c1 INTEGER
   ,c2 VARCHAR(9))
PRIMARY INDEX ( c1 );
```

Now consider the following function definition:

```
CREATE FUNCTION UDF_RAN(LowerBound INTEGER, UpperBound INTEGER)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
NOT DETERMINISTIC
EXTERNAL NAME 'JarUDF:UDFExample.udf_random';
```

The following SELECT statement includes a condition on the c1 index column that invokes the non-deterministic UDF called UDF\_RAN and results in an all-AMP operation:

```
SELECT *
FROM t1
WHERE c1 = UDF_RAN(1,12);
```

## Java UDF Execution Environment

Teradata supports two types of servers that can execute Java external routines (UDFs and external stored procedures). The type of server that executes a Java UDF depends on whether the CREATE FUNCTION or REPLACE FUNCTION statement specifies an EXTERNAL SECURITY clause.

IF the CREATE/REPLACE FUNCTION statement ...	THEN Teradata ...
specifies the EXTERNAL SECURITY clause	uses a separate secure server to execute the UDF under the authorization of a specific native operating system user established by a CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement.
does not specify the EXTERNAL SECURITY clause	executes the UDF as a thread of a hybrid server that runs under the authorization of the 'tdatuser' operating system user.

Whether a CREATE FUNCTION or REPLACE FUNCTION statement needs to specify the EXTERNAL SECURITY clause depends on the types of resources (if any) that the UDF needs to access during execution. For more information, see [Resource Access](#).

## Related Information

FOR more information on ...	SEE ...
Java server administration	<a href="#">Administration</a> .
CREATE FUNCTION and the EXTERNAL SECURITY clause	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the privileges associated with UDFs	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Function Name Overloading

UDFs support function name overloading: you can define several UDFs that have the same name but are different from each other in such a way to make each UDF unique.

The name that is overloaded is specified in the CREATE FUNCTION statement as the name to use to invoke the UDF from an SQL statement. You must specify a unique Java method for each UDF with the same name.

You can also overload functions of different classes (scalar, aggregate, and table). That is, a scalar function can have the same name as an aggregate function within the same database as long as the two functions are unique.

## Characteristics of a Unique Function

Functions that have the same name are unique if any of the following is true:

- The number of parameters is different.
- The parameter data types are distinct from each other.

## Relationship to CREATE FUNCTION Statement

The name that is overloaded is the name that immediately follows the CREATE FUNCTION keywords in the CREATE FUNCTION statement.

Each time you use CREATE FUNCTION to overload a function name, you must specify:

- A parameter list that satisfies the characteristics of a unique function
- A different Java method

Java methods can also be overloaded: you can write multiple methods with the same name but different parameters, located together in the same class and same installed JAR or ZIP file.

## Example: Overloaded Function

Consider the following overloaded Java methods called `get_random` in the `UDFExample` class:

```
public class UDFExample {

    ...

    public static int get_random( int some_value ) {
        ...
    }

    public static int get_random( java.sql.Date some_value ) {
        ...
    }

    ...

}
```

The method names are the same, but the data type of the *some\_value* parameter is distinct.

If the JAR file for the `UDFExample` class is called `UDFExample.jar`, the following statement registers `UDFExample.jar` and the *UDFExample* class with the database, and creates an SQL identifier called *JarUDF* for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!C:\udfsrc\UDFExample.jar','JarUDF',0);
```

Now consider the following function definitions that overload the scalar UDF name UDF\_RAN:

```
CREATE FUNCTION UDF_RAN(SomeValue INTEGER)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarUDF:UDFExample.get_random(int)';

CREATE FUNCTION UDF_RAN(SomeValue DATE)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarUDF:UDFExample.get_random(java.sql.Date)';
```

## Defining Functions that Use the TD\_ANYTYPE Type

Teradata provides a parameter data type called TD\_ANYTYPE that can accept any system-defined data type. You can specify TD\_ANYTYPE as a data type for:

- Input parameters in scalar, aggregate and table functions
- Result parameters in scalar and aggregate functions
- IN, INOUT, or OUT parameters in external stored procedures

You cannot use TD\_ANYTYPE:

- As the return type in table functions
- As input parameters and return value in UDMs because you cannot write UDMs in Java
- To represent UDT parameters because UDT parameters are not supported in Java routines

The parameter attributes and return type are determined at execution time based on the actual arguments passed to the routine.

The TD\_ANYTYPE parameter is mapped to java.lang.Object in the Java routine. You must specify java.lang.Object as the Java parameter type corresponding to the TD\_ANYTYPE parameter in the UDF.

## Example: Function with TD\_ANYTYPE Parameter and Return Type

Here is an example of how to declare a Java function that uses a TD\_ANYTYPE parameter and return type:

```
public static java.lang.Object get_smallest(java.lang.Object Obj1,
java.lang.Object Obj2) throws SQL Exception
```

The following shows the corresponding CREATE FUNCTION statement:

```
CREATE FUNCTION get_smallest(C1 TD_ANYTYPE, C2 TD_ANYTYPE)
  RETURNS TD_ANYTYPE
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.get_smallest';
```

You can also write the CREATE FUNCTION statement as:

```
CREATE FUNCTION get_smallest(C1 TD_ANYTYPE, C2 TD_ANYTYPE)
  RETURNS TD_ANYTYPE
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.get_smallest(java.lang.Object,
java.lang.Object) returns java.lang.Object';
```

For an example of a Java UDF that uses a TD\_ANYTYPE parameter, see [Java Function Using TD\\_ANYTYPE Parameters](#).

## The RETURNS and RETURNS STYLE Clauses

When invoking a scalar or aggregate UDF that is defined with a TD\_ANYTYPE result parameter, you can use the RETURNS *data type* or RETURNS STYLE *column expression* clauses to specify the desired return type. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

The RETURNS or RETURNS STYLE clause is not mandatory as long as the function also includes a TD\_ANYTYPE input parameter. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD\_ANYTYPE input argument is used to determine the return type of the TD\_ANYTYPE result parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.

You can use these clauses only with scalar and aggregate UDFs. You cannot use them with table functions. Also, you must enclose the UDF invocation in parenthesis if you use the RETURNS or RETURNS STYLE clauses.

## JAR and ZIP File Administration

One of the administrative tasks that you perform during the development of Java external routines (UDFs and external stored procedures) is JAR or ZIP file registration and distribution, as discussed in [Registering the JAR or ZIP File](#).

If the Java source code that implements an external routine changes, you must perform another administrative task: JAR or ZIP file replacement. To replace a previously registered JAR or ZIP file, use the SQLJ.REPLACE\_JAR external stored procedure. For details on using SQLJ.REPLACE\_JAR to replace a JAR or ZIP file, see [Replacing Registered JAR or ZIP Files](#).

For information on other administrative tasks that you perform during the development or maintenance of Java external routines, see [Administration](#).

## Using Java Reflection

If you use the Java Reflection API located in the java.lang.reflect package, you must take extra steps to be able to dynamically load and instantiate the modules that you use reflectively at runtime.

### Procedure

1. Copy the archive or class for the module that you will use on demand to a well known and uniform directory on each node.

For example, suppose you copy the archive file myclasses.jar to /java/MyClasses on each node of a Linux system.

2. Configure the JVM class path using the cufconfig utility to include the archive or class.

- a. Create a text file that contains the following text:

```
-cp <path1>
```

where <path1> is the path to the archive or class.

For example, you can create a text file called jvmenv.txt in the /tmp directory that contains the following:

```
-cp /java/MyClasses/myclasses.jar
```

- b. Set appropriate privileges for the text file:

```
chmod 666 jvmenv.txt
```

- c. Create another text file that contains the following text:

```
JavaEnvFile:<path2>
```

where <path2> is the path to and name of the text file you created previously.



For example, you can create a text file called `cuf.txt` that contains the following:

```
JavaEnvFile:/tmp/jvmenv.txt
```

- d. Use the `-f` option of the `cufconfig` utility, specifying the name of the file you created in the preceding step. For example:

```
cufconfig -f cuf.txt
```

3. Use the `-o` option of the `cufconfig` utility to verify your changes.

```
cufconfig -o
```

4. Restart the database using the following command for the changes to take effect:

```
tpareset -y
```

For details on `cufconfig` and `tpareset`, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Limitations

Unlike the JAR or ZIP archive files that you register with the database using the `SQLJ.INSTALL_JAR` external stored procedure, the archives that Vantage opens based on the preceding procedure are not replaceable and may not close until the database restarts.

# Java External Stored Procedures

In addition to writing external stored procedures in the C or C++ programming language, as described in [C/C++ External Stored Procedures](#), you can write external stored procedures in the Java programming language, install them on the database, and then use the SQL CALL statement to call them like other stored procedures.

Java external stored procedures can use JDBC to directly execute SQL statements.

This section uses the term *stored procedure* to refer to a stored procedure that is written using SQL statements and the term *external stored procedure* to refer to a stored procedure that is written using C, C++, or Java.

---

**Note:**

Customers using Vantage delivered as-a-service cannot create their own C++ and Java UDFs, UDMs, UDTs, or External Stored Procedures.

---

## Java Development Environment

The topics that the following sections discuss do not assume that you have any specific development environment beyond a Java SDK or JRE.

If you use Eclipse as a Java IDE, you can develop Java external stored procedures using the Teradata Plug-In for Eclipse, available from [Teradata Downloads](#). For details about creating Java external stored procedures and managing JAR files using the Teradata Plug-In for Eclipse, see *Teradata Plug-In for Eclipse Release Definition*, also available from Teradata Downloads.

## Overall Development Synopsis

### System Requirements

Before you can run Java external stored procedures, your system must meet certain requirements. For more information, see [System Requirements](#).

### Procedure

Here is a synopsis of the steps you take to develop, compile, install, and invoke a Java external stored procedure. You can find details for each step in subsequent sections.

1. Write, test, and debug the Java source code for the external stored procedure outside of the database and place the resulting class or classes in a JAR or ZIP file.
2. Call the SQLJ.INSTALL\_JAR external stored procedure to register the JAR or ZIP file and its classes with the database, providing an SQL identifier for the JAR or ZIP file.

- Determine the level of access to operating system services that the external stored procedure requires.

IF the external stored procedure ...	THEN ...
does not perform I/O or accesses operating system resources that ordinary operating system users have access to	the external stored procedure can run in protected execution mode as a separate process under 'tdatuser', a local operating system user that the database installation process creates.
requires access to specific operating system resources	use CREATE AUTHORIZATION or REPLACE AUTHORIZATION to create a context that identifies a native operating system user and allows the external stored procedure to access resources by using its own secure server under the authorization of that user.

The meaning of I/O as it applies to external stored procedures is any operating system call that requires the operating system to retain a resource context, such as for open files or other operating system services. Such resource usage usually returns a handle to the caller that is used to access the resource and must be released when finished using it.

- Use CREATE PROCEDURE or REPLACE PROCEDURE with options that provide specific information about the external stored procedure.

Option	Description
<ul style="list-style-type: none"> <li>• NO SQL (Default)</li> <li>• CONTAINS SQL</li> <li>• READS SQL DATA</li> <li>• MODIFIES SQL DATA</li> </ul>	Indicates whether the external stored procedure executes SQL statements and, if so, whether the statements read or modify SQL data in the database.
LANGUAGE JAVA	Identifies the source code language of the external stored procedure.
PARAMETER STYLE JAVA	Indicates the parameter style.
EXTERNAL NAME	Provides the registered name of the JAR file, Java class within the JAR file, and Java method within the class to execute when the external stored procedure is invoked in a CALL statement.
EXTERNAL SECURITY (Optional)	Associates execution of the external stored procedure with the context created by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement in the previous step.

- Test the external stored procedure in the database until you are satisfied it works correctly.
- Use GRANT to grant privileges to users who are authorized to use the external stored procedure.

## Related Information

FOR more information on ...	SEE ...
creating an external stored procedure	related topics in this document.

FOR more information on ...	SEE ...
debugging a Java external stored procedure	<a href="#">Debugging Using Trace Tables.</a>
registering a JAR file	<a href="#">Registering the JAR or ZIP File.</a>
<ul style="list-style-type: none"> <li>• CREATE PROCEDURE</li> <li>• REPLACE PROCEDURE</li> </ul>	<ul style="list-style-type: none"> <li>• <a href="#">Defining the SQL External Stored Procedure.</a></li> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
<ul style="list-style-type: none"> <li>• CREATE AUTHORIZATION</li> <li>• REPLACE AUTHORIZATION</li> </ul>	<ul style="list-style-type: none"> <li>• <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> <li>• <i>Teradata Vantage™ - Database Administration</i>, B035-1093.</li> </ul>
code examples for external stored procedures	<a href="#">External Stored Procedure Code Examples.</a>

## Class and Method Names

The name of the class and method in the source code follows the Java naming conventions.

When you use the CREATE PROCEDURE or REPLACE PROCEDURE statement to create an SQL identifier for the external stored procedure, you specify the name of the class and method in the EXTERNAL NAME clause. For more information, see [Specifying the Class Name, Method Name, and JAR Identifier](#).

## Parameter List

The list of parameters in a Java method for an external stored procedure includes the IN, OUT, and INOUT parameters that are specified when the external procedure appears in a CALL statement.

An external procedure can have 0 to 255 input/output parameters.

The Java data types that you use as input/output parameters map to SQL data types in the procedure definition (CREATE PROCEDURE or REPLACE PROCEDURE) and invocation (CALL).

### Note:

A Java external procedure cannot have an ARRAY type parameter where the base type is a nested structured UDT.

### Syntax

```
public class class_name {
    ...
    public static void method_name (
        [ input_parameter_specification [...] ]
```

```

    )
    {
        ...
    }
}

```

### ***input\_parameter\_specification***

```
type *input_parameter,
```

## **Syntax Elements**

### ***input\_parameter\_specification***

[Optional] Type and name of an input parameter in the CREATE PROCEDURE definition. Each input parameter in the definition must have a corresponding *input\_parameter\_specification*. The maximum number of input parameters is 128.

The *type* is a Java primitive or class that corresponds to the SQL data type of *input\_parameter*.

The maximum number of input/output parameters is 255.

## **Usage Notes**

### **Default Mapping Convention of Parameter Types**

The data types that you use in the parameter list of the Java method map to the SQL data types in the parameter list of the CREATE PROCEDURE or REPLACE PROCEDURE statement.

Consider the following CREATE PROCEDURE statement:

```

CREATE PROCEDURE NewRegionXSP
  (IN regionID INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP:region.newRegion';

```

The parameter list specifies that the SQL data type of *regionID* is INTEGER. The signature of the *newRegion* method that implements the external stored procedure looks like this:

```
public static void newRegion( int regionID ) { ... }
```

The default mapping convention is simple mapping, where SQL data types map to Java primitives. If no Java primitive can adequately map to an SQL type, then the default mapping convention is object mapping, where SQL data types map to Java classes.

Consider a `NewRegionXSP` external stored procedure that takes a `CHARACTER(30)` value:

```
CREATE PROCEDURE NewRegionXSP
  (IN regionID CHAR(30))
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP:region.newRegion';
```

Because the `CHARACTER` type does not map adequately to a Java primitive, the `CHARACTER` type maps to `java.lang.String`. The signature of the `newRegion` method that implements the external stored procedure looks like this:

```
public static void newRegion( String regionID ) { ... }
```

For details on how SQL data types map to Java data types, see [SQL Data Type Mapping](#).

## Overriding the Default Mapping of Parameters

For external stored procedures that allow parameters to pass in or return nulls, simple mapping to Java primitives is not appropriate because they cannot represent NULLs.

To override the default mapping, the `EXTERNAL NAME` clause in the `CREATE PROCEDURE` or `REPLACE PROCEDURE` statement must explicitly specify the mapping in the parameter list of the Java method.

## CLOB and BLOB Type Mapping

`CLOB` and `BLOB` SQL types in the parameter list of the `CREATE PROCEDURE` or `REPLACE PROCEDURE` statement map to `java.sql.Clob` and `java.sql.Blob` classes respectively.

The data access clause in the `CREATE PROCEDURE` or `REPLACE PROCEDURE` statement determines which implementing classes of `java.sql.Clob` and `java.sql.Blob` are used. (The data access clause indicates whether the external stored procedure executes SQL statements and, if so, whether the statements read or modify SQL data in the database.)

IF the SQL type is ...	AND the data access clause is ...	THEN the Java implementing class is ...
BLOB	NO SQL	<code>com.teradata.fnc.Blob</code> . For details, see <a href="#">com.teradata.fnc.Blob</a> .

IF the SQL type is ...	AND the data access clause is ...	THEN the Java implementing class is ...
	CONTAINS SQL	com.teradata.jdbc.jdbc_4.Blob. For details, see <i>Teradata JDBC Driver Reference</i> , available at <a href="https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html">https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html</a> .
	READS SQL DATA	
	MODIFIES SQL DATA	
CLOB	NO SQL	com.teradata.fnc.Clob. For details, see <a href="#">com.teradata.fnc.Clob</a> .
	CONTAINS SQL	com.teradata.jdbc.jdbc_4.Clob. For details, see <i>Teradata JDBC Driver Reference</i> , available at <a href="https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html">https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html</a> .
	READS SQL DATA	
	MODIFIES SQL DATA	

For details on the data access clause, see [Executing SQL in Java External Stored Procedures](#).

## Examples

### Example: Using Default Mapping of Parameter Types

Here is a code excerpt that shows how to implement a Java method for an external stored procedure that maps an SQL INTEGER type parameter to the Java int primitive:

```
public class region {

    public static void newRegion( int[] regionID )
    {
        regionID[0] += 1;
    }

    ...

}
```

If the JAR file for the region class is called `region.jar`, the following statement registers `region.jar` and the `region` class with the database, and creates an SQL identifier called `JarXSP` for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!C:\xspsrc\region.jar','JarXSP',0);
```

The corresponding CREATE PROCEDURE statement to define the external stored procedure looks like this :

```
CREATE PROCEDURE NewRegionXSP
  (INOUT regionID INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP:region.newRegion';
```

### Example: Overriding the Default Mapping of Parameter Types

Here is a code excerpt that shows how to implement a Java method for an external stored procedure that maps an SQL INTEGER type parameter to the java.lang.Integer class:

```
public class region {

    public static void newRegion( Integer[] regionID )
    {
        if (regionID[0] != null)
            regionID[0] = regionID[0].intValue() + 1;
    }

    ...

}
```

If the JAR file for the region class is called `region.jar`, the following statement registers `region.jar` and the `region` class with the database, and creates an SQL identifier called `JarXSP` for the JAR file:

```
CALL SQLJ.INSTALL_JAR('CJ!C:\xspsrc\region.jar','JarXSP',0);
```

The corresponding CREATE PROCEDURE statement to define the external stored procedure looks like this :

```
CREATE PROCEDURE NewRegionXSP
  (INOUT regionID INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP:region.newRegion(java.lang.Integer[])';
```



## External Stored Procedures That Use TD\_ANYTYPE Arguments

You can define external stored procedures with IN, INOUT, or OUT parameters that are of TD\_ANYTYPE data type. The TD\_ANYTYPE parameter data type can accept any system-defined data type. You cannot use TD\_ANYTYPE to represent UDT parameters because UDT parameters are not supported in Java routines.

The parameter attributes and return type are determined at execution time based on the actual arguments passed to the routine. For more information about the TD\_ANYTYPE type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

The TD\_ANYTYPE parameter is mapped to java.lang.Object in the Java routine. You must specify java.lang.Object as the Java parameter type corresponding to the TD\_ANYTYPE parameter in the external stored procedure.

### Example: External Stored Procedure with TD\_ANYTYPE Parameter Type

Here is an example of how to declare a Java external stored procedure that uses TD\_ANYTYPE parameter types:

```
public static void get_smallest(java.lang.Object C1, java.lang.Object[] C2)
```

The following shows the corresponding CREATE PROCEDURE statement:

```
CREATE PROCEDURE get_smallest(IN C1 TD_ANYTYPE, INOUT C2 TD_ANYTYPE)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME
  'UDF_JAR:UserDefinedFunctions.get_smallest';
```

### The RETURNS and RETURNS STYLE Clauses

When invoking an external stored procedure that is defined with a TD\_ANYTYPE OUT parameter, you can specify the RETURNS *data type* or RETURNS STYLE *column expression* clauses along with the OUT argument in the CALL statement to indicate the desired return type of the OUT parameter. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

The RETURNS or RETURNS STYLE clause is not mandatory as long as the procedure also includes a TD\_ANYTYPE input parameter. If you do not specify a RETURNS or RETURNS STYLE clause, then the data type of the first TD\_ANYTYPE IN or INOUT argument is used to determine the return type of the OUT parameter. For character types, if the character set is not specified as part of the data type, then the default character set is used.

The RETURNS and RETURNS STYLE clauses are only used to set the return type for a TD\_ANYTYPE OUT parameter. The data type of a TD\_ANYTYPE INOUT parameter is determined by the data type of the corresponding input argument.

## Class Fields

Class, or static, fields in Java classes may only represent constants and must be declared with the *final* modifier.

## Returning SQLSTATE Values

A Java external stored procedure can throw a java.lang.SQLException with an exception number and message that are returned to the database and used to set the SQLSTATE value and error message.

Valid values for the SQLState field of the SQLException are in the range 38U00 to 38U99.

Here is an example:

```
public class region {

    public static void newRegion( Integer[] regionID )
    throws SQLException
    {
        if (regionID[0] != null)
            regionID[0] = regionID[0].intValue() + 1;
        else
            throw new SQLException("Region ID not valid", "38U01");
    }

    ...

}
```

SQLException is the only valid exception that Java external stored procedures can throw.

## Executing SQL in Java External Stored Procedures

Java external stored procedures can use JDBC to establish a default connection to the database and directly execute SQL.

When an SQL statement within a Java external stored procedure accesses structured UDT, Period, or ARRAY type data, the values are returned based on the UDTTransformsOff, PeriodStructOn and ArrayTransformsOff flag set in the options parcel for the request.

This section provides highlights on using JDBC to execute SQL in Java external stored procedures, and identifies differences between accessing Vantage from a Java external stored procedure and

accessing the database from a Java application. For details on using Teradata Driver for the JDBC Interface to access the database, see *Teradata JDBC Driver Reference*, available at <https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html>.

## Usage Notes

### Establishing a Connection

Before a Java external stored procedure can execute SQL, it must establish a default connection to the database by passing "jdbc:default:connection" to the `java.sql.DriverManager.getConnection()` method.

The default connection sends SQL to the session that the external stored procedure is running in.

To successfully call `DriverManager.getConnection()` to access the Teradata JDBC Driver, the `CREATE PROCEDURE` or `REPLACE PROCEDURE` statement must specify one of the following data access clauses:

- CONTAINS SQL
- READS SQL DATA
- MODIFIES SQL DATA

For details on these data access clauses, see .

If you omit one of the preceding data access clauses, or if the `CREATE PROCEDURE` or `REPLACE PROCEDURE` statement specifies the `NO SQL` clause, then the Teradata JDBC Driver will not be available in the class path for the Java external stored procedure. An attempt to call the `DriverManager.getConnection` method to access the Teradata JDBC Driver results in an exception indicating "No suitable driver".

### Limiting the Types of SQL Statements that a Procedure Executes

The types of SQL statements that an external stored procedure (and stored procedure) can execute is restricted by the data access clause specified in the `CREATE PROCEDURE` or `REPLACE PROCEDURE` statement.

IF the <code>CREATE PROCEDURE</code> or <code>REPLACE PROCEDURE</code> specifies ...	THEN the external stored procedure ...
CONTAINS SQL	cannot read or modify SQL data in the database, but can execute SQL control statements such as <code>CALL</code> .
READS SQL DATA	cannot modify SQL data, but can execute statements such as <code>SELECT</code> that read SQL data.
MODIFIES SQL DATA	can execute SQL statements that read or modify SQL data.

An external stored procedure or stored procedure can only execute statements corresponding to the access clause of the most restrictive procedure that called it. For example, consider the following:

- A Java external stored procedure where the CREATE PROCEDURE statement specifies a data access clause of CONTAINS SQL.
- A stored procedure where the CREATE PROCEDURE statement specifies a data access clause of MODIFIES SQL DATA.

If the external stored procedure calls the stored procedure, the stored procedure can only execute control statements because the caller is already restricted by the CONTAINS SQL data access clause.

### Statements that a Procedure can Execute if Fired From a Trigger

An external stored procedure that is called when a trigger is fired can only execute SQL statements that are allowed as triggered action statements.

Attempts to execute other statements result in the database returning an exception to the external stored procedure. An external stored procedure that receives such an exception has an opportunity to post the error and close any external files or disconnect any connections it established. The only remaining course of action is for it to return.

IF the external stored procedure receives an exception and ...	THEN the triggering request is terminated and ...
returns with its own error by throwing an SQLException and setting the SQLState field to its own exception code	the original failure condition will not be known to the caller of the external stored procedure.
does not throw an SQLException	the system returns the original failure to the caller.
attempts to submit another request	the system generates a 7836 error: The XSP db.name submitted a request subsequent to receiving a trigger fail message.

For more details and a list of SQL statements that are allowed as triggered action statements, see the information about CREATE TRIGGER in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

### Restrictions

Java external stored procedures that execute SQL cannot be multithreaded.

## Examples

### Example: Establishing a Default Connection to Vantage

```
public class region {

    public static void getRegion(String[] data) throws SQLException
    {
        String sql = "SELECT Region FROM Sales WHERE ID = ";
        try {
```

```

    /* Establish default connection. */
    Connection con =
        DriverManager.getConnection( "jdbc:default:connection " );
    /* Execute the statement */
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery( sql + data[0] );
    rs.next();
    data[0] = rs.getString("Region");
    stmt.close();
}
catch (Exception e) {
    throw new SQLException(e.getMessage(),"38U01");
}
}

...
}

```

### Example: External Stored Procedure That Reads SQL Data

The following statement specifies the READS SQL DATA data access clause because the getRegion() method that implements the GetRegion external stored procedure (see ) executes a SELECT statement.

```

CREATE PROCEDURE GetRegion(INOUT Str VARCHAR(120))
LANGUAGE JAVA
READS SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP:region.getRegion';

```

## Related Information

FOR information on ...	SEE ...
CREATE PROCEDURE	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
REPLACE PROCEDURE	
CONTAINS SQL, READS SQL DATA, and MODIFIES SQL DATA data access clause	
using the Teradata Driver for the JDBC Interface	<i>Teradata JDBC Driver Reference</i> , available at <a href="https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html">https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html</a> .

FOR information on ...	SEE ...
UDTTransformsOff, PeriodStructOn and ArrayTransformsOff flags	<i>Teradata Vantage™ - Data Types and Literals</i> , B035-1143.

## Returning Dynamic Result Sets

If the CREATE PROCEDURE or REPLACE PROCEDURE statement for the external stored procedure specifies the DYNAMIC RESULT SETS clause, the external stored procedure can return result sets to the client application or to the caller of the external stored procedure (in addition to consuming the result sets itself) upon completion of the external stored procedure.

A *result set* is a set of rows that is the result of any of the following statements that the external stored procedure executes:

<ul style="list-style-type: none"> <li>• SELECT</li> <li>• HELP TABLE</li> <li>• HELP VIEW</li> <li>• HELP MACRO</li> </ul>	<ul style="list-style-type: none"> <li>• SHOW TABLE</li> <li>• SHOW VIEW</li> <li>• SHOW MACRO</li> <li>• COMMENT</li> </ul>
---	--

An external stored procedure can return up to 15 result sets, depending on the specification of the DYNAMIC RESULT SETS clause.

## Usage Notes

### Java Method Signature

The signature for a Java method that implements the external stored procedure includes the IN, INOUT, and OUT parameters, followed by *n* ResultSet[] output parameters, where *n* is the number of result sets specified in the DYNAMIC RESULT SETS clause.

### Implementation

To return a result set, a Java external stored procedure establishes a connection to Vantage using the default connection of the JDBC driver, executes one of the SQL statements that produces a result set, and uses Statement.getResultSet() to get the result set and return it using one of the ResultSet[] output parameters.

The Java external stored procedure must not close the Statement object that it uses to get the result set.

A Java external stored procedure can return as many result sets as specified by the DYNAMIC RESULT SETS clause of the CREATE PROCEDURE or REPLACE PROCEDURE statement. It is not mandatory that the external stored procedure return any result sets, however. A null or uninitialized result set output parameter indicates none is returned.

A Java external stored procedure can position each result set to a specific row. For example, an external stored procedure that returns three result sets can position one at the third row, one at the n-1 row, and one at the beginning.

## Restrictions

The following restrictions apply to an external stored procedure that returns result sets:

- A statement for which an external stored procedure returns a result set cannot be part of a multistatement request.
- Inline reading of BLOB or CLOB data is not supported from a result set.
- A calling Java external stored procedure cannot return the result set created by a called stored procedure to the caller of the Java external stored procedure. It can only consume the data. (For details on consuming result sets, see [Consuming Result Sets Created by Calling a Stored Procedure.](#))

## Performance and Resource Considerations

Creating result sets uses additional time and resources that you must consider. Because the environment (including character set and host) of the caller may differ from the environment of the external stored procedure, the database must generate two spool files for each statement that an external stored procedure creates a result set for.

## Example: Returning Two Result Sets

Here is an example of a method that returns two result sets:

```

/*****
DDL for the Java external stored procedure:

REPLACE PROCEDURE UsrCmd(  Command VARCHAR(120) )
  LANGUAGE JAVA
  READS SQL DATA
  PARAMETER STYLE JAVA
  DYNAMIC RESULT SETS 2
  EXTERNAL NAME 'jUdfRs:jUdfRsExamples.UsrCmd';

*****/

public class jUdfRsExamples {
  public static void UsrCmd(String Command,
                           ResultSet[] rs1,
                           ResultSet[] rs2) throws SQLException {
    Connection con =
      DriverManager.getConnection( "jdbc:default:connection" );
    Statement stmt =      con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,

```

```

        ResultSet.CONCUR_READ_ONLY);

    /* Execute user's command. */
    if (stmt.execute( Command )) {
        /* Return the result set. */
        rs1[0] = stmt.getResultSet();
    }
    if ( stmt.getMoreResults(Statement.KEEP_CURRENT_RESULT)) {
        /* Get a second result set if needed. */
        rs2[0] = stmt.getResultSet();
    }
}
...
}

```

## Consuming Result Sets Created by Calling a Stored Procedure

An external stored procedure can call a stored procedure that creates up to 15 dynamic result sets.

Here is a code example for a Java external stored procedure that calls a stored procedure called SQL\_SP\_W\_RS and consumes result sets created by the stored procedure:

```

/*****
DDL for the Java external stored procedure:

REPLACE PROCEDURE CountRegions( OUT RowsFound INTEGER )
LANGUAGE JAVA
READS SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'JarXSP:region.countRegion';
*****/

public class region {

    public static void countRegion(int[] rowcount) throws SQLException
    {
        /* Establish default connection. */
        Connection con =
            DriverManager.getConnection("jdbc:default:connection");

        /* Call SQL stored procedure that returns a result set */
        CallableStatement stmt =

```



```

        con.prepareCall("CALL SQL_SP_W_RS( ? )");
        stmt.setInt(1,1);

        /* Check that a result set was returned. */
        if (stmt.execute()) {
            /* Get returned result set. */
            ResultSet rs = stmt.getResultSet();
            rowcount[0]=0;
            /* Count rows in result set. */
            while (rs.next()) {
                rowcount[0]++;
            }
        }
    }

    ...
}

```

For details on how to return result sets from a stored procedure, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

## Teradata Application Classes

Teradata provides application classes that you can use for Java external stored procedure development. For example, you can use the `DbsInfo` class to get session information related to the current execution of a procedure.

The following table shows some of the categories of classes that Teradata provides.

Category	Use
Lob Access	For an external stored procedure that uses an SQL BLOB or CLOB type parameter and specifies a data access clause of NO SQL in the CREATE PROCEDURE or REPLACE PROCEDURE statement
Global Information	Returns session information related to the current execution of an external stored procedure
Query Band	Provides methods for external stored procedures that need to access query band information for a session, transaction, and/or profile
Trace	Let you get trace output for debugging purposes during external stored procedure development

For a list of all available classes and their descriptions, see [Java Application Classes](#).

## Compiling the Source Code

To compile the source code for your Java routine, you need javFnc.jar, the Teradata Java external stored procedure and UDF runtime library.

IF you compile the source code on ...	THEN ...
your client system	download the runtime library from <a href="#">Teradata Downloads</a> .
the database system	the runtime library is already available.

Add the path to the runtime library to your class path (or use the -classpath option of the Java compiler). For details on the location of the runtime library on a database system, see [Class Path](#).

For example, suppose you downloaded javFnc.jar to C:\java\_xsp on your Windows client. You can compile the source code in region.java as follows:

```
javac -classpath C:\java_xsp\javFnc.jar region.java
```

The version of the class file that you generate when you compile the Java source code must be compatible with the JRE on the database system.

For example, suppose the database uses JRE 1.8 and your client system uses JDK 8.0. To compile the source code in region.java on your client system, you can use the -target option of the Java compiler.

```
javac -target 1.8 -classpath C:\java_xsp\javFnc.jar region.java
```

To determine the version of the JRE on the database system, use the cufconfig utility to see where the JRE was installed. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Registering the JAR or ZIP File

After you write and test the Java source code for an external stored procedure, you place the resulting class or classes in an archive (JAR or ZIP) file.

Next, you call the SQLJ.INSTALL\_JAR external stored procedure to register the archive file and its classes with the database, specifying an SQL identifier for the archive file. The SQL identifier that you specify is the same name that you use later in the EXTERNAL NAME clause of the CREATE PROCEDURE or REPLACE PROCEDURE statement to define the external stored procedure.

For example, consider a JAR file called reports.jar on a Windows client in the directory C:\java\_xsp.

The following statements register the JAR file with the JXSP database and create an SQL identifier called Report\_JAR for the JAR file:

```
DATABASE JXSP;
CALL SQLJ.INSTALL_JAR('C:\java_xsp\reports.jar', 'Report_JAR', 0);
```

For details on how to register the archive file for a Java external stored procedure, see [Registering and Distributing JAR and ZIP Files for Java External Routines](#).

## Defining the SQL External Stored Procedure

After you register the JAR file containing the class or classes that implement the external stored procedure, you can define the SQL external stored procedure with the CREATE PROCEDURE or REPLACE PROCEDURE statement.

### Database for the External Stored Procedure

When you execute the CREATE PROCEDURE or REPLACE PROCEDURE statement to define an external stored procedure that is implemented by a class in the JAR file, you must define it in the same database in which the JAR file was registered.

### Specifying the Class Name, Method Name, and JAR Identifier

The CREATE PROCEDURE and REPLACE PROCEDURE statements provide the EXTERNAL NAME clause for specifying the identifier of the registered JAR file, and the class and method name in the JAR file that implements the external stored procedure.

Consider the following CREATE PROCEDURE statement:

```
CREATE PROCEDURE GetReport(INOUT Str VARCHAR(120))
  LANGUAGE JAVA
  READS SQL DATA
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'Report_JAR:report.getReport';
```

where:

THIS part of the EXTERNAL NAME clause ...	Specifies the ...
Report_JAR	identifier of the registered JAR file that was provided to the SQLJ.INSTALL_JAR procedure.
report	name of the class in the registered JAR file that implements the Java external stored procedure.
getReport	method name in the specified class that the database invokes when the external stored procedure is called in a CALL statement.

## Specifying the Type of Data Access

If the external stored procedure does not use JDBC to execute SQL, the CREATE PROCEDURE or REPLACE PROCEDURE statement can specify the NO SQL data access clause to indicate that the external stored procedure does not execute SQL.

If the external stored procedure executes SQL, the CREATE PROCEDURE or REPLACE PROCEDURE statement must specify one of the following data access clauses:

- CONTAINS SQL
- READS SQL DATA
- MODIFIES SQL DATA

For more information and an example, see [Executing SQL in Java External Stored Procedures](#).

## Default and Temporary Paths

To manage external stored procedures, Teradata uses default and temporary paths for external stored procedure creation and execution, including a temporary directory where external stored procedures are compiled.

For information, including the names of external stored procedure default and temporary paths, see *Teradata Vantage™ - Database Administration*, B035-1093.

## Related Information

FOR more information on ...	SEE ...
the CREATE PROCEDURE and REPLACE PROCEDURE statements	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
the privileges that apply to external stored procedures	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.

## Debugging Using Trace Tables

After an external stored procedure is installed in the database, debugging is limited.

The best practice is to develop and test the external stored procedure outside the database before you install it. You can use your own debugging tools to verify functionality.

If debugging outside the database is not sufficient, you can use trace tables to get trace diagnostic output.

## Overall Procedure

Here is a synopsis of the steps you take to debug a Java external stored procedure using trace tables.

### Note:

You can also use this procedure to debug Java UDFs.

1. Create a trace table using the CREATE GLOBAL TEMPORARY TRACE TABLE statement.

The first two columns of the trace table are used by the Teradata function trace subsystem. Any columns that are defined beyond the first two are available for an external stored procedure to use to write trace output during execution.

2. Enable the trace table for trace output using the SET SESSION FUNCTION TRACE statement.

You can specify an optional trace string that the external stored procedure can obtain during execution.

3. Invoke the external stored procedure.

4. Call `DbInfo.getTraceString()` in the external stored procedure to get the trace string that was specified in the SET SESSION FUNCTION TRACE statement.

You can use the value of the trace string to determine what to output to the trace table.

5. Call `DbInfo.traceWrite()` to write trace output to the columns of a trace table.

6. Use a SELECT statement to query the trace table and retrieve the trace output from the external stored procedure.

### Example: Debugging an External Stored Procedure Using a Trace Table

Consider the following statement that creates a trace table that defines one column, *Trace\_Output*, for an external stored procedure to write trace output to:

```
CREATE GLOBAL TEMPORARY TRACE TABLE XSP_Trace
  (vproc_ID      BYTE(2)
  ,Sequence      INTEGER
  ,Trace_Output  VARCHAR(256))
ON COMMIT PRESERVE ROWS;
```

The following code uses the value of the trace string to determine whether to write the value of the input argument to the trace table:

```
public static void debugJXSP(String[] Str) throws SQLException {
  try {
    /* If the trace string is set to 2, write the value */
    /* of the Str input argument to the trace table.    */
    if (Str[0]!=null && DbInfo.getTraceString().compareTo("2")==0) {
      DbInfo.traceWrite("Debug Info: " + Str[0]);
    }
    String[] x=null;
    /* The following is an exception. */
    if (x[0].compareTo(Str[0])==0) return;
  }
  catch (Throwable t) {
    StackTraceElement[] errs = t.getStackTrace();
    DbInfo.traceWrite(t.toString() + " " + t.getMessage());
  }
}
```

```

    for (int i=0;i<errs.length;i++)
        DbsInfo.traceWrite("In " + errs[i].getFileName() +
                           " at "+ errs[i].getLineNumber());
    throw new SQLException(t.getMessage(), "38U01");
}
}

```

The following statement enables trace output for table XSP\_Trace and sets the trace string to 2 so that the external stored procedure outputs the value of the input argument to the trace table:

```
SET SESSION FUNCTION TRACE USING '2' FOR TABLE XSP_Trace;
```

The SET SESSION FUNCTION TRACE statement disables any previously enabled trace tables for the session.

The following statement queries the trace table to retrieve the trace output from the external stored procedure:

```

SELECT Trace_Output
FROM XSP_Trace
ORDER BY Sequence;

```

## Related Information

FOR more information on ...	SEE ...
the DbsInfo.traceWrite() method	<a href="#">com.teradata.fnc.DbsInfo</a> .
the DbsInfo.getTraceString() method	
the TraceObj class	<a href="#">com.teradata.fnc.TraceObj</a> .
CREATE GLOBAL TEMPORARY TRACE TABLE	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
SET SESSION FUNCTION TRACE	

## Resolving UDF Server Setup Errors

When an external routine like an UDF, UDM, or external stored procedure is called, a UDF server process is acquired from the UDF server pool to execute the external routine. If there are no UDF server processes in the pool or if all of the processes in the pool are busy, then the system tries to start a new UDF server process for the request.

The startup of the new UDF server usually takes some time, especially if the UDF server is for executing Java external routines, or if the system is very busy. If the new UDF server cannot be started within the default time limit, the query that contains the UDF, UDM, or external procedure call is aborted, and you may

receive a 7583 error indicating that the UDF server setup encountered a problem. The system log may also record a 7820 error specifying that the UDF server could not stay up long enough for initialization.

If you are experiencing these errors, you can contact Teradata Support Center personnel to adjust the time limit allowed for starting a new UDF server process. For details, see the information about the Cufconfig utility in *Teradata Vantage™ - Database Utilities*, B035-1102.

## External Stored Procedure Invocation

Invoking an external stored procedure in a CALL statement is no different from invoking a stored procedure.

### Argument List

The arguments in the CALL statement must appear as comma-separated expressions in the same order as the list of parameters in the Java method.

The argument types of an external stored procedure in a CALL statement must either be compatible with the corresponding parameter declarations in the function declaration or must be types that are implicitly converted to the corresponding parameter types, according to implicit type conversion rules.

To pass an argument that is not compatible with the corresponding parameter type and is not implicitly converted by Vantage, the CALL statement must explicitly convert the argument to the proper type.

For information on data type conversions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

### Invoking External Stored Procedures with TD\_ANYTYPE OUT Parameters

When invoking an external stored procedure that is defined with a TD\_ANYTYPE OUT parameter, you can specify the RETURNS *data type* or RETURNS STYLE *column expression* clauses along with the OUT argument in the CALL statement to indicate the desired return type of the OUT parameter. The column expression can be any valid table or view column reference, and the return data type is determined based on the type of the column.

For detailed information, see [External Stored Procedures That Use TD\\_ANYTYPE Arguments](#).

### Nested Procedure Calls

When doing nested stored procedure calls only one of those procedures can be an external procedure.

## AT TIME ZONE Option for External Procedures

When you create an external procedure, the database stores the current session time zone for the procedure along with its definition to enable the SQL language elements in the procedure to execute in a consistent time zone and produce consistent results. However, time or timestamp data passed as an input parameter to the procedure still use the runtime session time zone rather than the creation time zone for the procedure.

The AT TIME ZONE option of the ALTER PROCEDURE statement enables you to reset the time zone for all of the SQL elements of external procedures when you recompile a procedure. The database then stores the newly specified time zone as the creation time zone for the procedure.

You can only specify AT TIME ZONE with the COMPILE option, and it must follow the COMPILE specification. If it does not, the database aborts the request and returns an error to the requestor. For details, see the information about ALTER PROCEDURE (External Form) in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Argument Behavior

### Truncation of Character String Arguments

The session transaction mode affects character string truncation.

IF the session transaction mode is ...	THEN an input character string that requires truncation is truncated ...
Teradata	without reporting an error. Truncation on Kanji1 character strings containing multibyte characters might result in truncation of one byte of the multibyte character.
ANSI	of excess pad characters without reporting an error. Truncation of other characters results in a truncation exception.

The normal truncation rules apply to a result string. For more information, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

### Behavior When Using NULL as a Literal Argument

To properly handle the NULL literal as an input argument, an external stored procedure cannot use the default mapping convention if the SQL data types in the parameter list of the CREATE PROCEDURE or REPLACE PROCEDURE statement map to Java primitives.

To override the default mapping and map SQL data types to Java classes that can handle the NULL literal as an input argument, the EXTERNAL NAME clause in the CREATE PROCEDURE or REPLACE PROCEDURE statement must explicitly specify the mapping in the parameter list of the Java method.

For details on how SQL data types map to Java data types, see [SQL Data Type Mapping](#). For an example that shows how to override the default mapping, see [Parameter List](#).

### Overflow and Numeric Arguments

To avoid numeric overflow conditions, the Java external stored procedure should define a DECIMAL or NUMERIC data type as big as it can handle.

If the assignment of the value of an input or output numeric argument would result in a loss of significant digits, a numeric overflow error is reported.

For example, consider an external stored procedure that takes a DECIMAL(2,0) argument:



```
CREATE PROCEDURE smldec( IN p1 DECIMAL(2,0) )
LANGUAGE JAVA
READS SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'NumJAR:NumClass.smldec';
```

Passing a number with a maximum of two digits is successful:

```
CALL smldec(99);
```

An attempt to pass a number larger than 99 or smaller than -99 would result in a loss of significant digits.

```
CALL smdec(100);
```

```
Failure 2616 Numeric overflow occurred during computation.
```

Any fractional numeric data that is passed or returned that does not fit as it is being assigned is rounded according to the Teradata rounding rules. For more information on rounding, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Data Type for INOUT Constant Arguments

The data type for an INOUT constant argument is governed by the data type of the value passed in, not what is defined. If the data type of the value passed in is smaller than the data type defined in the CREATE PROCEDURE or REPLACE PROCEDURE statement, and the external stored procedure returns a value larger than the maximum value of the data type for the value passed in, the system returns an overflow error.

For example, consider an external stored procedure that defines an INTEGER INOUT parameter:

```
CREATE PROCEDURE inout_example( INOUT p1 INTEGER )
LANGUAGE JAVA
READS SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'NumJAR:NumClass.inout_example';
```

If you call the external stored procedure with a constant input value that fits into a SMALLINT, the system returns an overflow error if the output value is larger than 32767, the maximum value of a SMALLINT:

```
CALL inout_example(1000);
```

## JAR and ZIP File Administration

One of the administrative tasks that you perform during the development of Java external stored procedures is archive (JAR or ZIP) file registration and distribution, as discussed in [Registering the JAR or ZIP File](#).

If the Java source code that implements an external stored procedure changes, you must perform another administrative task: archive file replacement. To replace a previously registered archive file, use the `SQLJ.REPLACE_JAR` external stored procedure. For details on using `SQLJ.REPLACE_JAR` to replace an archive file, see [Replacing Registered JAR or ZIP Files](#).

For information on other administrative tasks that you perform during the development or maintenance of Java external stored procedures, see [Administration](#).

## Using Java Reflection

If you use the Java Reflection API located in the `java.lang.reflect` package, you must take extra steps to be able to dynamically load and instantiate the modules that you use reflectively at runtime.

For more information, see [Using Java Reflection](#).

## Attempting to Exit the Java Virtual Machine

A Java external stored procedure must not call `System.exit()` or any similar methods. Any attempt to terminate or halt the currently running Java Virtual Machine by calling `System.exit()` or similar methods generates an error.

## R Table Operators

In advanced data analytics, statistical models are used to make predictions of future events based on current and historical data. R is a data analysis software and a programming language for statistical modeling and graphics.

Teradata provides in-database support for R program execution. Data stored in Teradata databases can be directly accessed by the R execution engine for analysis. R scripts are executed in parallel inside the database using the system table operator ExecR. R scripts can only be run as table operators and not as user-defined functions.

You can also write table operators that access the data in the database via R FNC functions and process the data using any of the data analysis tools available in R. R table operators include functionality such as the following:

- Support of multiple input streams

For example, you can store statistical models in the database and pass them to a scoring table operator as input streams.

- Support of contract functions and R FNC functions to set and get metadata

With this functionality, you can define polymorphic scripts. Also, a contract function can be used to precompute some intermediate result only once, and then pass this intermediate result to all AMPs using the contract context. This may result in performance improvements.

The table operator feature is a Teradata extension to the ANSI SQL:2011 standard.

## Installation of R Components and Packages

---

### Note:

Due to the complexity of installation, length of time and need for a database restart, Teradata strongly recommends enlisting Teradata Customer Services to install the R Components and Packages. Please contact your Teradata representative to order this service.

---

In Vantage systems, R can be used to execute R scripts through the SCRIPT table operator or the ExecR table operator.

To use R with the SCRIPT table operator, your system administrator must install the R interpreter. For details about SCRIPT, see the Table Operators section in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

To use R with the ExecR table operator, your system administrator must install the following components:

- The R interpreter
- The Fortran compiler

- The udfGPL library

Furthermore, the DIP script DipRTblOp must be executed to create the ExecR table operator. The rest of the present section has more information about the R interpreter and Fortran packages. For more information about installing the udfGPL library and creating the ExecR table operator, see [Procedure to Enable R Functionality with ExecR](#).

Together with the R interpreter, R add-on packages can be used to apply a variety of statistical techniques to analyze data stored in the Advanced SQL Engine. Once installed, a new package (pkg) can be used within an R script by including the following statement: `library(pkg)`.

In Vantage systems,

- The R interpreter is provided through the teradata-R package
- Selected R add-on packages are available through the teradata-R-addons package (for a listing of the add-on titles that are bundled in a given release, see the README file of that release)
- The Fortran compiler comprises of the packages:
  - gcc-fortran, gcc43-fortran, libgfortran43 (in SLES 11 Service Pack 3 systems)
  - gcc-fortran, gcc48-fortran, libgfortran3 (in SLES 12 Service Pack 3 systems)

The teradata-R interpreter is a prerequisite for the teradata-R-addons package. See [R and Python Support in Teradata Vantage](#) for R and Python support in-database in Vantage.

## Using ExecR to Execute R Scripts and Table Operators

Support of languages for table operators can be classified as first and second class. First class support of a language includes registration of table operators in the database. For example, C is supported as a first class language. C operators are registered in the database using DDL statements such as CREATE FUNCTION.

R is supported as a second class language. R table operators are not registered in the database and are executed by the ExecR system table operator in Teradata. ExecR is created by the DipRTblOp DIP script, and it resides in the td\_sysgpl database. ExecR can be executed only in protected mode because R is not thread safe.

The R code for the contract function and the table operator is passed to ExecR in USING clauses. During execution of ExecR, the contract and operator are interpreted by the R interpreter.

If the R code for the contract function and operator is less than 64 KB, you can pass the contract and operator as a string in the USING clauses.

```
SELECT *
FROM TD_SYSGPL.ExecR (
  ON (select * from tab1)
  [Hash By / Partition By / Local Order By / DIMENSION clauses]
  [ON ...]
  USING Contract(' <R contract function>' )
  Operator(' <R operator>' )
```

```

        [ Other USING clauses]
        ...
    ) AS D;

```

If the R code for the contract function and operator is greater than 64 KB, but smaller than 4 MB, you can pass the code as LOBs in the USING clauses. For example, consider a table named prohtable with attributes id (int) and rcode (clob). The following query interprets rcode with id 1 as the contract and rcode with id 2 as the operator.

```

SELECT *
FROM TD_SYSGPL.ExecR (
    ON (select * from tab1)
    [Hash By / Partition By / Local Order By / DIMENSION clauses]
    [ON ...]
    USING Contract(select rcode from prohtable where id=1)
        Operator(select rcode from prohtable where id=2)
        [ Other USING clauses]
    ...
) AS D;

```

You can omit the contract function if you are using a RETURNS clause to define the output column definitions. The RETURNS clause can specify an output table or the column names and their corresponding types.

---

**Note:**

You must specify either the contract function or a RETURNS clause.

---

If you specify a RETURNS clause, it must come before the USING clause inside the ExecR call. The following shows an example of specifying a RETURNS clause instead of the contract function.

```

SELECT *
FROM TD_SYSGPL.ExecR (
    ON (select * from t1)
    RETURNS (a integer, b smallint, c bigint, d byteint, e float, f real)
    USING
        Operator ('
            library(tdr);
            ...
        ')
) AS D1;

```

## Supported Character Sets for R Programs

The R interpreter can only parse programs written in UTF8 or the ASCII character set; therefore, R programs must be written in the LATIN character set. Other character sets, such as UNICODE, cannot be parsed successfully by R.

The table operators containing R programs should be written in the LATIN character set. Data stored in table columns that is referenced by the program must also be in the LATIN character set. Even LOB data has to be in the LATIN character set. Otherwise errors may be thrown or unexpected results may be produced.

## Memory Limitation

R table operators use the udfGPL secure server. The GPLUDFServerMemSize field in the cufconfig GDO (Globally Distributed Object) is used to limit the memory of the udfGPL server. This limit is set to 3.5 GB by default, but you can set the limit from 0 to 3.5 GB. To use the maximum available memory, set GPLUDFServerMemSize to 0; however, a nonzero value is recommended so that there is a limit to the amount of memory that can be used.

For information about the cufconfig utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

If you set a limit over 2 GB, you get a warning message:

```
Warning: Although the memory limit is set, it is very high. Your system has
high risk of running out of virtual memory. Please consider to set it to a
lower value.
```

If you set a limit over 3.5 GB, you get an "out of range" error:

```
Out of range error. Field GPLUDFServerMemSize: 0 - 3758096384
```

The maximum amount of memory that can be used by an R table operator is determined by the GPLUDFServerMemSize limit. It includes user data and other metadata used by R for various maintenance activities. Any attempt to use more than the limit results in evaluation errors while running the program. The error may be caused by the memory request from either user program, R internal calls, or operating system calls.

The udfGPL server may seem to consume more memory than the specified limit. This is expected because various resources are required to start and run the user programs. The GPLUDFServerMemSize limit is in addition to whatever the server uses. For example, assume that GPLUDFServerMemSize is set to 32 MB. When the udfGPL server starts, it may need 103 MB. The table operator would have 32 MB in addition to the 103 MB for usage. Therefore, in total, about 135 MB of memory may be consumed by the user program or operator. The memory may be split evenly between different streams (input and output) of the table operator when required depending on the memory usage requirements of the operator.

A GPL server might not be started if the total of GPLUDFServerMemSize plus the memory space for the R server itself after initialization is bigger than UINT\_MAX at runtime. In this case, you will get an error:

```
Error 7583 The secure mode processes had a set up error.
```

Note that when an R table operator is executed, there is some memory held by the R interpreter. R generally does not reclaim all memory used by the program objects. It often tries to reclaim such unused memory by running garbage collector automatically. Therefore, in the next run of another R table operator, the available

memory may be less than the `GPLUDFServerMemSize` limit. You can overcome this by forcefully releasing the memory used by different objects in the environment. For example, `rm(list=ls())`.

## keepLog USING Clause

The `keepLog USING` clause can be used with the operator to produce log files. Any print statements in the R program appear in the log. The log files are stored in `/home/tdatuser/`.

## R Table Operator Example

```
CREATE TABLE tab1(col1 INTEGER,col2 INTEGER)PRIMARY INDEX(COL1);
INSERT INTO tab1 VALUES(1, 20);
INSERT INTO tab1 VALUES(2, 30);
INSERT INTO tab1 VALUES(3, 40);
INSERT INTO tab1 VALUES(4, 50);

SELECT *
FROM TD_SYSGPL.ExecR (
  on (select * from tab1)
  hash by col1
  local order by col2
  using contract('library(tdr);
    stream<-0;
    direction<-"R";

    incols<-tdr.GetColDef(stream, direction);
    tdr.SetOutputColDef(stream,incols)')
  operator('library(tdr);
    stream<-0;
    direction<-"R";
    direction1<-"W";
    options<-0;
    inHandle<-tdr.Open(direction, stream, options);

    print(inHandle);
    outHandle<-tdr.Open(direction1, stream, options);

    print(outHandle);
    colcount <- tdr.GetColCount(stream , direction);

    colcount <- colcount -1 ;
    while(tdr.Read(inHandle)== 0)
    {
```

```

        for( index in 0:colcount )
        {
            att <- tdr.GetAttributeByNdx( inHandle , index , NULL);
            tdr.SetAttributeByNdx(outHandle , index , att, NULL);
        };
        tdr.Write(outHandle);
    };
    tdr.Close(inHandle);
    tdr.Close(outHandle);')
) as d1;

```

Results:

col1	col2
1	20
4	50
2	30
3	40

## R Table Operator Example: Echo Example

The example shows an R program that echoes a table it reads from input.

### Prerequisites

The following is required to run the example:

- R 3.2.1 or later installed
- Run DipRTblOp, if not previously run

### Echo Example

```

create table test(i int, j int);

insert into test values(10,1);
insert into test values(20,2);
insert into test values(30,3);
insert into test values(40,4);
insert into test values(50,5);

sel * from test;

sel * from td_sysgpl.execrc (
    ON (sel * from test)

```



```

using
keepLog(1)
contract
(
  'library(tdr);
  on_clause_input_stream <- 0;
  on_clause_output_stream <- 0;
  direction <- "R";

  incols <- tdr.GetColDef(on_clause_input_stream, direction);
  tdr.SetOutputColDef(on_clause_output_stream, incols);'
)
operator
(
  'library(tdr);

  streamin <- 0;
  streamout <- 0;
  read_direction <- "R";
  write_direction <- "W";

  options <- 0;
  inHandle <- tdr.Open(read_direction, streamin, options);
  outHandle <- tdr.Open(write_direction, streamout, options);

  numcols <- tdr.GetColCount(streamin, read_direction);

  while (tdr.Read(inHandle) == 0)
  {
    lapply( 0: (numcols - 1),
      function(index){
        att <- tdr.GetAttributeByNdx(inHandle, index, NULL);
        tdr.SetAttributeByNdx(outHandle, index, att, NULL);
      });

    tdr.Write(outHandle);
  };

  tdr.Close(inHandle);
  tdr.Close(outHandle);'
)
) as Rexp;

```

Result:

i	j
-----	-----
20	2
10	1
40	4
30	3
50	5

### Explanation of Echo Example

For this query the ON clause is kept simple. There is no Hash By, Partition By, nor Order By specified. Thus none of the rows are redistributed and are read as is on the AMPs for input to the Operator code. Because the first column values are unique, we can specify a hash by i (or j) to ensure that each AMP (on a four AMP system) gets at least one row.

The keepLog key is specified with a value of 1 to write stdout to files in /home/tdatuser. There will be one log file for the Contract and N log files for the Operator (one for each AMP) on an N AMP system.

In both the Contract and the Operator, when using tdr.GetColDef and tdr.Open respectively, streamin is used to reference input from an ON clause by position. Integers from 0 to 15 can be specified because up to 16 ON clauses can be used. In the Contract, the variable is called *on\_clause\_input\_stream*.

The Contract is executed by a single thread and can store information to be used by the AMPs when they process the Operator code. The main purpose of the Contract is to set up the output column definitions. In this case, the output column definitions are simply the same as the input.

Because each of the rows in the test table are on different AMPs and because the Operator code gets executed by each AMP in parallel, the input for each AMP will typically vary. The consequence of this can be seen in the while loop. The while loop keeps calling tdr.Read on the input handle until it reaches the end of the input stream.

The tdr.Read function reads a row at a time from the input stream. AMPs that have no input rows to process will just close the input and output handles. They don't write anything to the output stream. For AMPs that do have input, the tdr.Read function will read the data into the input handle which can then be used to write output.

This latter case is seen inside the while loop in the example. After the row is read into the input handle, each column value is read from the input handle using tdr.GetAttributeByNdx and then subsequently written to the corresponding column of the output handle using tdr.SetAttributeByNdx whose column types were defined in the contract.

## R Table Operator Use Case: Grouping Using K Means

This use case uses the iris dataset to group species together using K Means. The entire analysis shown can be done in one query, but it is broken into smaller steps to highlight ExecR's capabilities.

The following tasks are performed:

- Load the iris dataset into a table.

- Create the training and test tables and labels. Save the species predictions using knn.
- Use the predictions previously computed to calculate a ratio.

## Load the Dataset

This example shows an INSERT ... SELECT query. The result set from the ExecR query gets inserted into the iris\_data table. The example creates a table that has columns that match the columns in the iris dataset in R.

The tdr.TblWrite function is used to write whole data frames to the output stream. The iris dataset is a built-in dataset recognized in R.

```
create multiset table iris_data(sepal_length float, sepal_width float,
petal_length float, petal_width float, species int);

insert iris_data
sel * from td_sysgpl.execr (
  ON (sel * from iris_data) dimension
  ON (sel 1) PARTITION BY 1
  using
  keepLog(1)
  contract
  (
    'library(tdr);
    on_clause_input_stream <- 0;
    direction <- "R";

    incols <- tdr.GetColDef(on_clause_input_stream, direction);
    tdr.SetOutputColDef(on_clause_input_stream, incols);'
  )
  operator
  (
    'library(tdr);

    stream <- 0;
    options <- 0;
    write_direction <- "W";

    outHandle <- tdr.Open(write_direction, stream, options);
    tdr.TblWrite(outHandle, iris);

    tdr.Close(outHandle);'
  )
) as Rexp;
```

Select from the `iris_data` table to show the iris data loaded:

```
sel * from iris_data;
```

Result: The following shows partial results from the SELECT statement.

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	1
2	4.9	3.0	1.4	0.2	1
3	4.7	3.2	1.3	0.2	1
4	4.6	3.1	1.5	0.2	1
5	5.0	3.6	1.4	0.2	1
6	5.4	3.9	1.7	0.4	1
[...]					

Because the Operator code is executed in parallel, we need to redistribute the rows so that the Operator code makes sense for the input it gets. In our case, we want to use a fraction of all the iris data to create a training set and then use the rest for the test set.

```
sel count(*) from iris_data;
```

```
*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.
```

```
Count(*)
-----
      150
```

The ExecR query below counts the number of rows in the `iris_data` table on each AMP. The contract will give the definition for the output. The output is one column with the count of the number of rows expressed as an integer. Each row in the output is the result for one AMP.

The Operator code that is run on each AMP reads in the table's rows a chunk at a time and creates them into a data frame using `tdr.TblRead`. The `nrow` function in R is used to count the number of rows of the data frame.

```
SELECT * FROM TD_SYSGPL.ExecR (
  ON (sel * from iris_data)
  USING
    keeplog(1)
    Contract
    (
      'library(tdr)
      streamno_out <- 0
      coltype_integer <-list(datatype="INTEGER_DT", bytesize="SIZEOF_INTEGER")
```

```

    coldef <- list(count = coltype_integer)
    tdr.SetOutputColDef(streamno_out, coldef)'
)

Operator
(
'library(tdr)
streamno_in <- 0
streamno_out <- 0

handle_in <- tdr.Open("R", streamno_in, 0)
handle_out <- tdr.Open("W", streamno_out, 0)

rowsread<- 0
buffSize <- as.integer(12*1024)
someRows <- tdr.TblRead(handle_in, buffSize)

while( nrow(someRows) > 0 )
{
  rowsread<- rowsread + nrow(someRows)
  someRows <- tdr.TblRead(handle_in, buffSize)
}

tdr.Close(handle_in)
dat_export <- data.frame(
  count = as.integer(rowsread)
)

tdr.TblWrite(handle_out, dat_export)
tdr.Close(handle_out)'
) as Rexp;

*** Query completed. 4 rows found. One column returned.
*** Total elapsed time was 1 second.

count
-----
29
36
48
37

```

This count shows the distribution of rows that each AMP (on a four AMP system) has for its input. If we were to create our model in the Operator now, we'd get four different models trained and tested on different subsets of the same data. We want to generate one model based on a single training set and test set from all of the data.

To keep this use case simple, we will get all the data to be processed onto one AMP. To do this, use the following ON clause combination:

```
ON (sel 1) PARTITION BY 1
```

```
ON (sel * from iris_data) DIMENSION
```

This will force the iris\_data rows onto exactly one AMP. The ON clause specified with the DIMENSION option copies the table for every partition on which the table operator operates. In this case, since we are partitioning by a column with just one value, only one copy of the table will get created.

```
SELECT * FROM TD_SYSGPL.ExecR (
  ON (sel 1) PARTITION BY 1
  ON (sel * from iris_data) DIMENSION
  USING
    keeplog(1)
    Contract
    (
      'library(tdr)
      streamno_out <- 0
      coltype_integer <- list(datatype="INTEGER_DT", bytesize="SIZEOF_INTEGER")
      coldef <- list(count = coltype_integer)
      tdr.SetOutputColDef(streamno_out, coldef)'
    )

  Operator
  (
    'library(tdr)
    streamno_in <- 1
    streamno_out <- 0

    handle_in <- tdr.Open("R", streamno_in, 0)
    handle_out <- tdr.Open("W", streamno_out, 0)

    rowsread<- 0
    buffSize <- as.integer(12*1024)
    someRows <- tdr.TblRead(handle_in, buffSize)

    while( nrow(someRows) > 0 )
    {
      rowsread<- rowsread + nrow(someRows)
```

```

        someRows <- tdr.TblRead(handle_in, buffSize)
    }

    someRows <- tdr.TblRead(handle_in, buffSize)
    rowsread <- rowsread + nrow(someRows)

    tdr.Close(handle_in)
    dat_export <- data.frame(
        count = as.integer(rowsread)
    )

    tdr.TblWrite(handle_out, dat_export)
    tdr.Close(handle_out)'
)
) as Rexp
;

*** Query completed. One row found. One column returned.
*** Total elapsed time was 1 second.

      count
-----
      150

```

Note, because there is another ON clause, the input stream must be changed to reference the correct input. So, the value of streamno\_in changed to 1 in the Operator to match with the iris\_data input.

### Create the Training and Test Tables and Labels and Save with knn

In this example we read in all of the data created in the previous example then split it into a training set and a test set. We save the correct Species labels for comparison with the predicted labels in a data frame. This example uses serialize to save a data frame of predictions and correct labels for comparison later. The serialized data is saved as a blob in the predictions table.

```

create table predictions (id int, model blob);

insert predictions
sel * from td_sysgpl.execr (
    ON (sel 1) PARTITION BY 1
    ON (sel * from iris_data) DIMENSION
using
keepLog(1)
contract
(

```

```

'library(tdr);

on_clause_output_stream <- 0;

integer <- list(datatype="INTEGER_DT", bytesize="SIZEOF_INTEGER");
binary_lob <- tdr.Blob(1024);

coldefs <- list( key = integer, model = binary_lob );
tdr.SetOutputColDef(on_clause_output_stream, coldefs);'
)
operator
(
'library(tdr);
library(class);

streamin <- 1;
streamout <- 0;
read_direction <- "R";
write_direction <- "W";

options <- 0;
inHandle <- tdr.Open(read_direction, streamin, options);
outHandle <- tdr.Open(write_direction, streamout, options);

### Read Data ###

buffSize <- as.integer(16*1024)
data <- tdr.TblRead(inHandle, buffSize);

### Process ###

set.seed(2017);
ind <- sample(2, nrow(data), replace = TRUE, prob = c(0.67, 0.33));
training <- data[ind == 1, 1:4];
test <- data[ind == 2, 1:4];

train_labels <- data[ind == 1, 5];
test_labels <- data[ind == 2, 5];

prediction <- knn(train = training, test = test, cl = train_labels, k=3);
result <- data.frame(prediction, test_labels);

### Write Data ###

```



```

    saved <- serialize(result, NULL);
    locator <- tdr.LobCol2Loc(streamout, 1);
    tdr.LobAppend( locator, saved);

    tdr.SetAttributeByNdx(outHandle, streamout, list(value = 1L,
nullindicator=0), NULL);

    tdr.Write(outHandle);

    tdr.Close(inHandle);
    tdr.Close(outHandle);'
  )
) as Rexp;

```

Note, that the buffer size allocated to read the table is much larger than the size of the input, so we only need to call `tdr.TblRead` once. In general, the input should be read into data frames using `nrow` as in the previous examples.

Once the input is read into the data frame we create a vector of labels using `sample`, which are used to index the data frame for training and test data. We use two-thirds for the training set and one-third for the test. We then use the `knn` function provided by the `class` library in R to generate the predictions. The prediction and the `test_labels` are put into a data frame and serialized. Finally, they are written to the predictions table.

### Use the Predictions to Calculate a Ratio

In this example, we read the predictions data frame that we saved and use it to compare predicted labels with the actual labels. This scenario is useful for saving computed models for reuse or other intermediate results into data frames to use later.

```

sel count(*) from iris_data;

sel * from predictions;

sel * from td_sysgpl.execrc (
  ON (sel 1) PARTITION BY 1
  ON (sel * from predictions) DIMENSION
  using
  keepLog(1)
  contract
  (
    'library(tdr);

    on_clause_output_stream <- 0;

    integer <- list(datatype="INTEGER_DT", bytesize="SIZEOF_INTEGER");

```

```

real <- list(datatype="REAL_DT", bytesize="SIZEOF_REAL");

coldefs <- list( modelno = integer, right_ratio = real );
tdr.SetOutputColDef(on_clause_output_stream, coldefs);'
)
operator
(
'library(tdr);

streamin <- 1;
streamout <- 0;
read_direction <- "R";
write_direction <- "W";

options <- 0;
inHandle <- tdr.Open(read_direction, streamin, options);
outHandle <- tdr.Open(write_direction, streamout, options);

### Read Saved Prediction ###

tdr.Read(inHandle);
index <- tdr.GetAttributeByNdx(inHandle, 0L, NULL);
locator <- tdr.GetAttributeByNdx(inHandle, 1L, NULL);

inLob <- tdr.LobOpen_CL(locator, 0, 0);
model <- tdr.LobRead(inLob$contextID, inLob$LOBlen);

raw <- unlist(model$buffer);
prediction <- unserialize(raw);
tdr.Close(inHandle);

### Process Ratio ###

total <- nrow(prediction);
correct <- Filter( function(x){ x },
                  prediction[[1]] == prediction[[2]] );

result <- length(correct) / total;

### Write Data ###

tdr.SetAttributeByNdx(outHandle, 0L, list(value = index$value,
nullindicator = 0), NULL);
tdr.SetAttributeByNdx(outHandle, 1L, list(value = result, nullindicator =

```

```

0), NULL);

    tdr.Write(outHandle);
    tdr.Close(outHandle);'
)
) as Rexp;

```

## Related Information

For more information about table operators, see the following:

- *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146. See information about the SELECT table operator.
- *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

## R FNC Functions

An R table operator typically accesses and modifies data stored in the database to perform an assigned task.

The following sections describe FNC functions available to R table operator developers. These functions provide an interface for R scripts to access and update data and metadata from the database.

### Note:

R reads integer values as double. Therefore, you must cast the value to integer using `as.integer()` or other method. Otherwise you may get a data type mismatch error.

The following categories of functions are provided.

### Functions for Data Manipulation

For table operators, the rows in the input and output streams are accessed following an iterator model such as the following:

1. Open the input and output streams.
2. Sequentially read the rows in the input stream.
3. Directly access the columns in the current input row.
4. Update the columns in the current output row.
5. Sequentially write the output rows.
6. Close the streams.

The following functions support the iterator model and are used to manipulate the input and output streams. These functions allow you to access and modify data in the database. Functions are also available to set and retrieve the format specification of input and output streams.

As with C protected mode table operators, a buffer is used for reading and writing rows. When a read row is issued, the system reads an entire buffer of rows so that the next time a read row is issued, it retrieves the

row from the buffer. A new buffer is read when the end of the buffer is reached. Similarly, a buffer is used for writes. The size of the buffer is configurable and can be set in the `cufconfig` `GDO(CServerMemSize)`.

Function	Description
<a href="#">tdr.Open</a>	Opens an input or output stream.
<a href="#">tdr.Close</a>	Closes an input or output stream.
<a href="#">tdr.Read</a>	Reads a row from an input stream.
<a href="#">tdr.Write</a>	Writes a row to the output stream.
<a href="#">tdr.TblRead</a>	Reads an input stream into an R data frame and returns the data frame.
<a href="#">tdr.TblWrite</a>	Writes an R data frame to the output stream.
<a href="#">tdr.GetAttributeByNdx</a>	Retrieves an individual attribute from the current input stream row.
<a href="#">tdr.SetAttributeByNdx</a>	Sets the value of an individual attribute in the current output stream row.
<a href="#">tdr.GetStreamCount</a>	Retrieves the number of input and output streams.
<a href="#">tdr.GetFormat</a>	Retrieves information about the format of a specified stream.
<a href="#">tdr.SetFormat</a>	Sets an attribute of the format for a specified stream.
<a href="#">tdr.DisableCoGroup</a>	Disables the cogroup functionality for table operators that handle multiple input streams.

## Functions for Metadata Manipulation

For table operators, some metadata passed during query invocation can be accessed from the contract function and the operator. This metadata can be set in the contract function if it was not specified as part of the query. In addition, a contract function can set a context that can be accessed during execution of the operator.

The following functions are used to access and set this metadata and the contract context.

Function	Description
<a href="#">tdr.GetColCount</a>	Retrieves the number of columns in a stream.
<a href="#">tdr.GetColDef</a>	Retrieves the schema of the input or output stream.
<a href="#">tdr.SetOutputColDef</a>	Sets the schema of the output stream.
<a href="#">tdr.GetCountHashBy</a>	Retrieves the number of columns in the HASH BY clause for a specified input stream.
<a href="#">tdr.GetHashByDef</a>	Retrieves the columns in the HASH BY clause for a specified input stream.
<a href="#">tdr.SetHashByDef</a>	Sets the HASH BY specification for a specified input stream.
<a href="#">tdr.GetCountLocalOrderBy</a>	Retrieves the number of columns in the LOCAL ORDER BY clause for a specified input stream.

Function	Description
<a href="#">tdr.GetLocalOrderByDef</a>	Retrieves the columns in the LOCAL ORDER BY clause of a specified input stream, including the column ordering and a NULLS FIRST/NULLS LAST indicator.
<a href="#">tdr.SetLocalOrderByDef</a>	Sets the ordering specification for a specified input stream.
<a href="#">tdr.GetContractLength</a>	Retrieves the length of the contract context.
<a href="#">tdr.GetContractDef</a>	Retrieves the contract context.
<a href="#">tdr.SetContractDef</a>	Sets the contract context.
<a href="#">tdr.GetAsClauseName</a>	Retrieves the alias name (AS name) and associated length for a specified stream.
<a href="#">tdr.IsDimension</a>	Checks whether a stream is a DIMENSION input stream.
<a href="#">tdr.SetError</a>	Sets an error code and message to be displayed on the console when the query invoking the table operator is executed and an error is encountered.

### Functions to Access USING Clause Parameters

You can use USING clauses to pass parameters as key-value pairs to the contract function and table operator. The following functions are used to access these pairs in the contract function and table operator.

Function	Description
<a href="#">tdr.GetCustomKeyCount</a>	Retrieves the number of keys in the USING clauses.
<a href="#">tdr.GetCustomValuesOf</a>	Retrieves the values associated with a specified key.
<a href="#">tdr.GetCustomValuesAt</a>	Retrieves the values associated with a key at the specified index.
<a href="#">tdr.GetCustomKeyAt</a>	Retrieves the key at the specified index.

### Functions for LOB Manipulation

The following functions are used to access and modify LOBs from R table operators. Using these functions, R table operators can read LOBs from input streams and create LOBs for output streams.

Function	Description
<a href="#">tdr.LobOpen_CL</a>	Opens a LOB for reading.
<a href="#">tdr.LobClose</a>	Closes an input LOB.
<a href="#">tdr.GetLobLength</a>	Retrieves the length of an input LOB.
<a href="#">tdr.LobRead</a>	Reads a LOB.
<a href="#">tdr.LobCol2Loc</a>	Computes the output LOB locator needed to append data to a LOB in the current output row.

Function	Description
<a href="#">tdr.LobAppend</a>	Appends raw data to the end of the output LOB associated with a specified output locator.

## Functions For Setting Column Definitions

The following functions are used to generate lists for each of the supported data types. You can then pass these lists to the `tdr.SetOutputColDef` function. This allows you to define the lists in a quick and efficient manner so that you will not have to explicitly define each and every list used in calls to `tdr.SetOutputColDef`.

Some of the functions simply return the default attributes required for a specific data type. Other functions indicate certain properties about the data type, such as the length or character type, based on the parameters passed to the function.

### Character and CLOB Data Types

Function	Description
<a href="#">tdr.Char</a>	Retrieves the column definition for the CHAR_DT data type.
<a href="#">tdr.VarChar</a>	Retrieves the column definition for the VARCHAR_DT data type.
<a href="#">tdr.Clob</a>	Retrieves the column definition for the CLOB_REFERENCE_DT data type.

### Byte and BLOB Data Types

Function	Description
<a href="#">tdr.Byte</a>	Retrieves the column definition for the BYTE_DT data type.
<a href="#">tdr.VarByte</a>	Retrieves the column definition for the VARBYTE_DT data type.
<a href="#">tdr.Blob</a>	Retrieves the column definition for the BLOB_REFERENCE_DT data type.

### Numeric Data Types

Function	Description
<a href="#">tdr.ByteInt</a>	Retrieves the column definition for the BYTEINT_DT data type.
<a href="#">tdr.SmallInt</a>	Retrieves the column definition for the SMALLINT_DT data type.
<a href="#">tdr.Integer</a>	Retrieves the column definition for the INTEGER_DT data type.
<a href="#">tdr.BigInt</a>	Retrieves the column definition for the BIGINT_DT data type.
<a href="#">tdr.Decimal1</a>	Retrieves the column definition for the DECIMAL1_DT data type.
<a href="#">tdr.Decimal2</a>	Retrieves the column definition for the DECIMAL2_DT data type.
<a href="#">tdr.Decimal4</a>	Retrieves the column definition for the DECIMAL4_DT data type.
<a href="#">tdr.Decimal8</a>	Retrieves the column definition for the DECIMAL8_DT data type.

Function	Description
<a href="#">tdr.Decimal16</a>	Retrieves the column definition for the DECIMAL16_DT data type.
<a href="#">tdr.Float</a>	Retrieves the column definition for the REAL_DT data type.
<a href="#">tdr.Real</a>	Retrieves the column definition for the REAL_DT data type.

### DateTime and Interval Data Types

Function	Description
<a href="#">tdr.Date</a>	Retrieves the column definition for the DATE_DT data type.
<a href="#">tdr.Time</a>	Retrieves the column definition for the TIME_DT data type.
<a href="#">tdr.TimeWTZ</a>	Retrieves the column definition for the TIME_WTZ_DT data type.
<a href="#">tdr.Timestamp</a>	Retrieves the column definition for the TIMESTAMP_DT data type.
<a href="#">tdr.TimestampWTZ</a>	Retrieves the column definition for the TIMESTAMP_WTZ_DT data type.
<a href="#">tdr.IntervalYear</a>	Retrieves the column definition for the INTERVAL_YEAR_DT data type.
<a href="#">tdr.IntervalYTM</a>	Retrieves the column definition for the INTERVAL_YTM_DT data type.
<a href="#">tdr.IntervalMonth</a>	Retrieves the column definition for the INTERVAL_MONTH_DT data type.
<a href="#">tdr.IntervalDay</a>	Retrieves the column definition for the INTERVAL_DAY_DT data type.
<a href="#">tdr.IntervalDTH</a>	Retrieves the column definition for the INTERVAL_DTH_DT data type.
<a href="#">tdr.IntervalDTM</a>	Retrieves the column definition for the INTERVAL_DTM_DT data type.
<a href="#">tdr.IntervalDTS</a>	Retrieves the column definition for the INTERVAL_DTS_DT data type.
<a href="#">tdr.IntervalHour</a>	Retrieves the column definition for the INTERVAL_HOUR_DT data type.
<a href="#">tdr.IntervalHTM</a>	Retrieves the column definition for the INTERVAL_HTM_DT data type.
<a href="#">tdr.IntervalHTS</a>	Retrieves the column definition for the INTERVAL_HTS_DT data type.
<a href="#">tdr.IntervalMinute</a>	Retrieves the column definition for the INTERVAL_MINUTE_DT data type.
<a href="#">tdr.IntervalMTS</a>	Retrieves the column definition for the INTERVAL_MTS_DT data type.
<a href="#">tdr.IntervalSecond</a>	Retrieves the column definition for the INTERVAL_SECOND_DT data type.

### Additional Functions for Developing R Table Operators

The following additional functions may also be useful when developing R table operators.

Function	Description
<a href="#">tdr.ampinfo</a>	Returns information related to the AMP and node where the table operator is running.

Function	Description
<a href="#">tdr.dbsinfo</a>	Returns information related to the current running table operator or contract function, such as user account and user name.
<a href="#">tdr.getnodedata</a>	Returns node IDs and AMP IDs for all online AMP vprocs, allowing table operators to configure themselves to run on specific AMPs.
<a href="#">tdr.tracestring</a>	Returns the current setting of the trace string that was set up with the SET SESSION FUNCTION TRACE statement.
<a href="#">tdr.tracewrite</a>	Writes trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement.

## Prerequisite

Before calling these functions, the TDR package must be loaded by including the following statement in the R script.

```
library(tdr)
```

## Terminology

R types do not include scalar types. Scalar types such as integers are typically represented as a unit vector of the given type (integer). In the following sections, unit vectors of a type are specified by giving only the type. For example, integer and character denote integer and character unit vectors, respectively. Vectors with more than one element are specified as type vector, for example, integer vector.

## Mapping Between R and C Data Types

The data types of input and output parameters of R FNC functions conform to typical usage of R data types. However, R FNC functions are implemented using existing C FNC functions. Therefore, wrappers are used to convert data from R data types to C data types and vice versa. The mapping between R and C data types is shown in the following table.

R	C
integer	<ul style="list-style-type: none"> <li>signed char</li> <li>signed/unsigned short</li> <li>signed/unsigned int</li> <li>signed/unsigned long</li> </ul>
real	<ul style="list-style-type: none"> <li>float</li> <li>double</li> <li>long long</li> </ul>
character	char, when used for characters and strings



R	C
raw	<ul style="list-style-type: none"> <li>pointers</li> <li>unsigned char (byte)</li> </ul>
vector	array
list	struct

## Mapping Between Teradata Data Types and R Types

The following table shows the mapping from Teradata attribute types into R types. This mapping is used by the `tdr.GetAttributeByNdx` and `tdr.SetAttributeByNdx` functions to convert an attribute value and type from its Teradata (or C) representation into its R representation and vice versa.

Teradata Data Types	R
<ul style="list-style-type: none"> <li>CHAR</li> <li>VARCHAR</li> </ul>	Character vector
<ul style="list-style-type: none"> <li>BYTE</li> <li>VARBYTE</li> </ul>	Raw vector
<ul style="list-style-type: none"> <li>BYTEINT</li> <li>SMALLINT</li> <li>INTEGER</li> </ul>	Integer
<ul style="list-style-type: none"> <li>BIGINT</li> <li>REAL</li> <li>DECIMAL1, DECIMAL2, DECIMAL4</li> </ul>	Real
DECIMAL8, DECIMAL16	Raw vector
DATE	Date (number of days since 1/1/1970)
TIME	List with 3 elements: <ul style="list-style-type: none"> <li>seconds (Real unit vector)</li> <li>hour (Integer unit vector)</li> <li>minutes (Integer unit vector)</li> </ul> These three elements are the members of the C structure used to represent the TIME data type.
<ul style="list-style-type: none"> <li>TIME_WTZ</li> <li>TIMESTAMP</li> <li>TIMESTAMP_WTZ</li> </ul>	List with elements in the C structure used to represent the type in Teradata. The types of elements follow the mapping in <a href="#">Mapping Between R and C Data Types</a> .
<ul style="list-style-type: none"> <li>INTERVAL_YEAR</li> <li>INTERVAL_MONTH</li> </ul>	Integer

Teradata Data Types	R
<ul style="list-style-type: none"> <li>• INTERVAL_DAY</li> <li>• INTERVAL_HOUR</li> <li>• INTERVAL_MINUTE</li> </ul>	
<ul style="list-style-type: none"> <li>• INTERVAL_YTM</li> <li>• INTERVALH</li> <li>• INTERVALM</li> <li>• INTERVALS</li> <li>• INTERVAL_HTM</li> <li>• INTERVAL_HTS</li> <li>• INTERVAL_MTS</li> <li>• INTERVAL_SECOND</li> </ul>	List with elements in the C structure used to represent the type in Teradata. The types of elements follow the mapping in <a href="#">Mapping Between R and C Data Types</a> .
<ul style="list-style-type: none"> <li>• BLOB_REFERENCE</li> <li>• CLOB_REFERENCE</li> </ul>	Raw vector

This mapping was defined based on the following criteria:

- The R data types are available in all R environments
- The data conversion is lossless

The selected R types are part of the basic R and not specific packages that need to be installed. You can convert the returned data to the representation used by a package of your choice.

The XML and JSON data types are treated as LOBs.

The following data types are not supported:

- ARRAY
- Geospatial
- NUMBER
- Period
- UDT

## tdr.ampinfo

Returns the following information:

- Node ID
- AMP ID
- Lowest AMP ID in the node

### Syntax

```
tdr.ampinfo()
```

## tdr.BigInt

Returns an R list representing the column definition for an instance of the BIGINT\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.BigInt()
```

### Example: Retrieve a Column Definition For the BIGINT\_DT Type

```
coldef <- list(col= tdr.BigInt());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Blob

Returns an R list representing the column definition for an instance of the BLOB\_REFERENCE\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Blob(length)
```

### Syntax Elements

#### *length*

Parameter type: integer

The number of bytes to allocate for the column.

### Example: Retrieve a Column Definition For the BLOB\_REFERENCE\_DT Type

```
coldef <- list(col= tdr.Blob(512));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Byte

Returns an R list representing the column definition for an instance of the BYTE\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Byte(size)
```

## Syntax Elements

### *size*

Parameter type: integer

The number of bytes allotted to this column.

### Example: Retrieve a Column Definition For the BYTE\_DT Type

```
coldef <- list(col= tdr.Byte(100));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.ByteInt

Returns an R list representing the column definition for an instance of the BYTEINT\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.ByteInt()
```

### Example: Retrieve a Column Definition For the BYTEINT\_DT Type

```
coldef <- list(col= tdr.ByteInt());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Char

Returns an R list representing the column definition for an instance of the CHAR\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Char(size, chartype)
```

## Syntax Elements

### *size*

Parameter type: integer

The number of characters or bytes allotted to this column.

### *chartype*

Parameter type: string

The character set for the CHAR column being defined.

### Example: Retrieve a Column Definition For the CHAR\_DT Type

```
coldef <- list(col= tdr.Char(15,"UNICODE_CT"));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Clob

Returns an R list representing the column definition for an instance of the CLOB\_REFERENCE\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Clob(chartype, length)
```

### Syntax Elements

#### *chartype*

Parameter type: string

The character set for the CLOB column being defined.

#### *length*

Parameter type: integer

The number of characters to allocate for the column.

### Example: Retrieve a Column Definition For the CLOB\_REFERENCE\_DT Type

```
coldef <- list(col= tdr.Clob("LATIN_CT", 2097088000));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Close

Closes an input or output stream.

Returns an integer value of 0 if the operation was successful.

### Syntax

```
tdr.Close( handle )
```

## Syntax Elements

### *handle*

Parameter type: raw vector

The handle of the input or output stream that you want to close. This handle was previously returned by the `tdr.Open` function when you opened the stream.

## Usage Notes

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Closing an Input Stream

# Open the input stream.

```
library(tdr);
stream <- 0;
options <- 0;
direction <- "R";
inHandle <- tdr.Open(direction, stream, options);
```

# Close the input stream.

```
tdr.Close( inHandle );
```

## tdr.Date

Returns an R list representing the column definition for an instance of the `DATE_DT` data type with the given attributes, to use with the `tdr.SetOutputColDef` function.

### Syntax

```
tdr.Date()
```

### Example: Retrieve a Column Definition For the DATE\_DT Type

```
coldef <- list(col= tdr.Date());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.dbsinfo

Returns the following information:

- User account
- User name
- User ID
- Statement number
- Host ID
- Session number
- Request number

### Syntax

```
tdr.dbsinfo()
```

## tdr.Decimal1

Returns an R list representing the column definition for an instance of the DECIMAL1\_DT data type with the given attributes, to use with the `tdr.SetOutputColDef` function.

### Syntax

```
tdr.Decimal1(totaldigit, fracdigit)
```

### Syntax Elements

#### *totaldigit*

Parameter type: integer

The total number of digits in the decimal value.

#### *fracdigit*

Parameter type: integer

The number of fractional digits in the decimal value.

### Example: Retrieve a Column Definition For the DECIMAL1\_DT Type

```
coldef <- list(col= tdr.Decimal1(2,1));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Decimal2

Returns an R list representing the column definition for an instance of the DECIMAL2\_DT data type with the given attributes, to use with the `tdr.SetOutputColDef` function.

**Syntax**

```
tdr.Decimal2(totaldigit, fracdigit)
```

**Syntax Elements*****totaldigit***

Parameter type: integer

The total number of digits in the decimal value.

***fracdigit***

Parameter type: integer

The number of fractional digits in the decimal value.

**Example: Retrieve a Column Definition For the DECIMAL2\_DT Type**

```
coldef <- list(col= tdr.Decimal2(4,1));
tdr.SetOutputColDef(stream, coldef);
```

**tdr.Decimal4**

Returns an R list representing the column definition for an instance of the DECIMAL4\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.Decimal4(totaldigit, fracdigit)
```

**Syntax Elements*****totaldigit***

Parameter type: integer

The total number of digits in the decimal value.

***fracdigit***

Parameter type: integer

The number of fractional digits in the decimal value.



**Example: Retrieve a Column Definition For the DECIMAL4\_DT Type**

```
coldef <- list(col= tdr.Decimal4(9,6));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Decimal8

Returns an R list representing the column definition for an instance of the DECIMAL8\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.Decimal8(totaldigit, fracdigit)
```

**Syntax Elements*****totaldigit***

Parameter type: integer

The total number of digits in the decimal value.

***fracdigit***

Parameter type: integer

The number of fractional digits in the decimal value.

**Example: Retrieve a Column Definition For the DECIMAL8\_DT Type**

```
coldef <- list(col= tdr.Decimal8(18,9));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Decimal16

Returns an R list representing the column definition for an instance of the DECIMAL16\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.Decimal16(totaldigit, fracdigit)
```

**Syntax Elements*****totaldigit***

Parameter type: integer

The total number of digits in the decimal value.

### ***fracdigit***

Parameter type: integer

The number of fractional digits in the decimal value.

### **Example: Retrieve a Column Definition For the DECIMAL16\_DT Type**

```
coldef <- list(col= tdr.Decimal16(38,24));
tdr.SetOutputColDef(stream, coldef);
```

## **tdr.DisableCoGroup**

Disables the cogroup functionality for table operators that handle multiple input streams.

### **Syntax**

```
tdr.DisableCoGroup()
```

### **Usage Notes**

You can call `tdr.DisableCoGroup` in the `contract` function to turn off the cogroup functionality.

#### **Note:**

If cogroup is disabled, table operators that handle multiple input streams may return different results on systems with different configurations where the number of AMPs differ. To get consistent results on different configurations, cogroup must be enabled.

For more information about cogroups, see *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210.

## **tdr.Float**

Returns an R list representing the column definition for an instance of the `REAL_DT` data type with the given attributes, to use with the `tdr.SetOutputColDef` function.

### **Syntax**

```
tdr.Float()
```

**Example: tdr.Float - Retrieve a Column Definition For the REAL\_DT Type**

```
coldef <- list(col= tdr.Float());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.GetAsClauseName

Returns the name from the AS name clause for a specified stream in UNICODE, if it exists; returns null otherwise..

**Syntax**

```
tdr.GetAsClauseName(streamno,direction)
```

**Syntax Elements*****streamno***

Parameter type: integer

The input or output stream number.

***direction***

Parameter type: character

Valid values are as follows:

- "R" (input)
- "W" (output)

**Example: Get the Name From the AS *name* Clause**

```
stream <- 0;
direction <- "R";
name <- tdr.GetAsClauseName(stream,direction);
print(name);
```

The following shows the result from the print statement:

```
$value
[1] "abc"
```

## tdr.GetAttributeByNdx

Returns one of the following attributes from the current input stream row:

- A list representing the selected attribute value. This list consists of the following three elements:
  - *value*
  - *nullindicator*
  - *length*

If *nullindicator* has a value of -1, the attribute is NULL.

The type of *value* depends on the attribute type. See [Mapping Between Teradata Data Types and R Types](#) for a mapping from Teradata attribute types into R types.

- NULL if there is an error.

Error conditions include the following:

- The handle is associated with an output stream.
- The input stream is not open.
- The index is not valid.
- The *coldef* list contains an invalid data type or an incomplete set of attributes for that data type.
- The attribute is not of type list.
- This function was called from the contract function.

## Syntax

```
tdr.GetAttributeByNdx( handle, index, coldef)
```

## Syntax Elements

### *handle*

Parameter type: raw vector

The handle of the input stream returned by the `tdr.Open` function.

### *index*

Parameter type: integer

The index of an attribute. The valid range is from 0 to  $n-1$ , where  $n$  is the number of attributes in the stream. Index 0 indicates the first attribute.

### *coldef*

Parameter type: list

The schema of a stream. This schema is a list with the number of columns and the definition of each column. Each definition includes the column information for the data type of that specific column. This is the return value of a call to the `tdr.GetColDef` function.

If *coldef* is NULL, the function will retrieve the *coldef* information.

## Usage Notes

The *coldef* parameter can be used to improve performance. By passing the *coldef* information to the function, the function does not have to retrieve it on each call. However, you can optionally pass NULL instead and the function will retrieve the *coldef* information by itself.

Before you call this function, you must call the `tdr.Open` function to open the input stream. Then pass the handle returned from `tdr.Open` as an argument to this function.

---

### Note:

This function is valid only if called from the table operator.

---

### Example: Retrieve the First Attribute From Rows in the Input Stream

# Open the input stream.

```
library(tdr);
stream <- 0;
options <- 0;
direction <- "R";
inHandle <- tdr.Open(direction, stream, options);
```

# A data frame is created with the retrieved attributes.

```
mydataframe <- data.frame();
while (tdr.Read(inHandle) == 0) {
  att <- tdr.GetAttributeByNdx(inHandle, 0, NULL);
  newrow <- data.frame(att$value);
  mydataframe <- rbind(mydataframe, newrow);
}
```

Consider an input stream with a row whose first attribute is of type INTEGER with a value of 11. Object *att* for this row would be the following list:

```
$length
[1] 4
$value
[1] 11
$nullIndicator
[1] 0
```

Attribute *value* is an integer.

The previous example can optionally be written to pass the *coldef* information to the `tdr.GetAttributeByNdx` function. This would improve performance.

```
mydataframe <- data.frame();
coldef <- tdr.GetColDef(streamno, direction);
while (tdr.Read(inHandle) == 0) {
  att <- tdr.GetAttributeByNdx(inHandle, 0, coldef);
  newrow <- data.frame(att$value);
  mydataframe <- rbind(mydataframe, newrow);
}
```

## tdr.GetColCount

Returns an integer representing the number of columns in a stream.

### Syntax

```
tdr.GetColCount( streamno, direction )
```

### Syntax Elements

#### *streamno*

Parameter type: integer

The stream number.

#### *direction*

Parameter type: character

Valid values are as follows:

- "R" (input)
- "W" (output)

### Example: Get the Number of Columns in an Input Stream

```
stream <- 0;
direction <- "R";
count <- tdr.GetColCount( stream, direction );
```

## tdr.GetColDef

Returns a list with the schema of the specified stream, including the definition of each column, which includes the following information:

- Name
- Type

Valid data type values are defined by the `dtype_en` enumeration in the `sqltypes_td.h` header file.

- Size (in bytes)
- Character type for textual fields:
  - 1 for LATIN
  - 2 for UNICODE

## Syntax

```
tdr.GetColDef( streamno, direction )
```

## Syntax Elements

### *streamno*

Parameter type: integer

The stream number.

### *direction*

Parameter type: character

Valid values are as follows:

- "R" (input)
- "W" (output)

## Example: Get the Column Definitions of an Input Stream

```
stream <- 0;
direction <- "R";
incols <- tdr.GetColDef(stream, direction);
print(incols);
```

The list *incols* contains the following for a stream with three attributes (COL1 int, COL2 real, COL3 varchar(30)):

```
$COL1
$COL1$datatype
[1] 9

$COL1$bytesize
[1] 4

$COL2
$COL2$datatype
```

```
[1] 10

$COL2$bytesize
[1] 8

$COL3
$COL3$datatype
[1] 2

$COL3$charset
[1] 1

$COL3$size.length
[1] 30
```

**Note:**

You can get the number of columns in the stream by calling the length function:

```
length(incols)
```

## tdr.GetContractDef

Returns the contract context as a raw vector.

**Syntax**

```
tdr.GetContractDef()
```

**Example: Get the Contract Context**

The following statements set the contract context:

```
myctx <- list( division="appliances", codes=c(10, 34) );
myctxS <- serialize( myctx, NULL );
tdr.SetContractDef( myctxS );
```

The following statement retrieves the previous contract context in raw form and transforms it to its original form:

```
myctx <- unserialize(tdr.GetContractDef());
```



## tdr.GetContractLength

Returns an integer representing the number of bytes in the contract context.

### Syntax

```
tdr.GetContractLength()
```

### Example: Get the Length of the Contract Context

The following statement prints the number of bytes in the contract context:

```
print(tdr.GetContractLength())
```

## tdr.GetCountHashBy

Returns an integer representing the number of columns in the HASH BY clause for a specified input stream.

### Syntax

```
tdr.GetCountHashBy( streamno )
```

### Syntax Elements

#### *streamno*

Parameter type: integer

The input stream number.

### Example: Get the Number of Columns in the HASH BY Clause

This example gets the number of columns in the HASH BY clause for input stream 0.

```
print(tdr.GetCountHashBy( 0 ));
```

## tdr.GetCountLocalOrderBy

Returns an integer representing the number of columns in the LOCAL ORDER BY clause for a specified input stream.

### Syntax

```
tdr.GetCountLocalOrderBy( streamno )
```

## Syntax Elements

### *streamno*

Parameter type: integer

The input stream number.

### Example: Get the Number of Columns in the LOCAL ORDER BY Clause

This example gets the number of columns in the LOCAL ORDER BY clause for input stream 0.

```
print( tdr.GetLocalOrderHashBy( 0));
```

## tdr.GetCustomKeyAt

Returns a character string representing the key at the specified index.

If an error occurs, the function returns NULL.

### Syntax

```
tdr.GetCustomKeyAt(index)
```

## Syntax Elements

### *index*

Parameter type: integer

The function will search for the key at this index.

Keys are indexed in the order that they appear in the query, starting at 0.

## tdr.GetCustomKeyCount

Returns an integer representing the number of key-value pairs in the USING clause. The key count includes the operator and contract clauses.

### Syntax

```
tdr.GetCustomKeyCount()
```

## tdr.GetCustomValuesAt

Returns a vector with the values associated with the key at the specified index.

If an error occurs, the function returns NULL.

## Syntax

```
tdr.GetCustomValuesAt(index)
```

## Syntax Elements

### *index*

Parameter type: integer

The function will search for the key at this index.

Keys are indexed in the order that they appear in the query, starting at 0.

## tdr.GetCustomValuesOf

Returns a vector with the values associated with the specified key.

If an error occurs, the function returns NULL.

## Syntax

```
tdr.GetCustomValuesOf(key)
```

## Syntax Elements

### *key*

Parameter type: character

The function will search for this key.

## tdr.GetFormat

Returns an integer that indicates the record format attribute for the given stream as follows:

- 1 for IndicData with row separator sentinels
- 2 for IndicData with no row or partition separator sentinels

## Syntax

```
tdr.GetFormat( attribute, streamno, direction )
```

## Syntax Elements

### *attribute*

Parameter type: character

The format attribute to be retrieved.

Currently, the only valid value for *attribute* is "RECFMT", which defines the input/output record format.

### *streamno*

Parameter type: integer

The stream number.

### *direction*

Parameter type: character

Valid values are as follows:

- "R" (input)
- "W" (output)

## Usage Notes

This function can be called from the table operator as well as the contract function.

### Example: Print the Record Format of an Input Stream

The following statement prints the record format of input stream 0:

```
print(tdr.GetFormat("RECFMT", 0, "R"))
```

## tdr.GetHashByDef

Returns the column names in the HASH BY clause for a specified input stream.

## Syntax

```
tdr.GetHashByDef( streamno )
```

## Syntax Elements

### *streamno*

Parameter type: integer

The input stream number.

### Example: Get the Columns in the HASH BY Clause

Consider the following HASH BY clause for input stream 0:

```
HASH BY col1, col3
```

The following statement retrieves the columns in the HASH BY clause and prints them.

```
print( tdr.GetHashByDef( 0));
```

The output of the print statement is as follows:

```
$Num_Columns
[1] 2

$ColumnNames
[1] "col1" "col3"
```

## tdr.GetLobLength

Returns a real value representing the number of bytes in the specified LOB.

### Syntax

```
tdr.GetLobLength(locator)
```

### Syntax Elements

#### *locator*

Parameter type: integer

See how this function retrieves the length of an input LOB.

### Usage Notes

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Get the Length of an Input LOB

```
# Open the input stream
inHandle <- tdr.Open("R", 0, 0);
```

```
# Get the 5th attribute in the current row (LOB locator)
att5 <- tdr.GetAttributeByNdx(inHandle, 4, NULL);

# Get the length of the input LOB
length <- tdr.GetLobLength(att5);
```

## tdr.GetLocalOrderByDef

Returns a list of triplets, one for each column in the LOCAL ORDER BY clause of the specified input stream. Each triplet is a vector that consists of the following:

- A column name
- The ordering of the column ("A" = ascending or "D" = descending)
- An indication of whether NULLS FIRST ("F") or NULLS LAST ("L") was specified for the column

### Syntax

```
tdr.GetLocalOrderByDef( streamno )
```

### Syntax Elements

#### *streamno*

Parameter type: integer

The input stream number.

### Example: Get the Columns in the LOCAL ORDER BY Clause

Consider the following LOCAL ORDER BY clause for input stream 0:

```
LOCAL ORDER BY col1 ASC NULLS FIRST, col3 DESC NULLS LAST
```

The following statement retrieves information about the columns in the LOCAL ORDER BY clause and prints it.

```
print( tdr.GetLocalOrderByDef( 0 ));
```

The output of the print statement is as follows:

```
[[1]]
[1] "col1" "A"    "F"
```

```
[[2]]
[1] "col2" "D"    "L"
```

## tdr.getnodedata

Returns a list of node ID - AMP ID pairs for all online AMP vprocs, allowing table operators to configure themselves to run on specific AMPs.

### Syntax

```
tdr.getnodedata()
```

## tdr.GetStreamCount

Returns a list with two values:

- The number of input streams
- The number of output streams

### Syntax

```
tdr.GetStreamCount()
```

### Example: Get the Number of Input and Output Streams

```
x <- tdr.GetStreamCount()
```

Object *x* will be the following list:

```
$inputCount
[1] 1

$outputCount
[1] 1
```

## tdr.Integer

Returns an R list representing the column definition for an instance of the INTEGER\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Integer()
```

**Example: Retrieve a Column Definition For the INTEGER\_DT Type**

```
coldef <- list(col= tdr.Integer());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalDay

Returns an R list representing the column definition for an instance of the INTERVAL\_DAY\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalDay()
```

**Example: Retrieve a Column Definition For the INTERVAL\_DAY\_DT Type**

```
coldef <- list(col= tdr.IntervalDay());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalDTH

Returns an R list representing the column definition for an instance of the INTERVAL\_DTH\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalDTH()
```

**Example: Retrieve a Column Definition For the INTERVAL\_DTH\_DT Type**

```
coldef <- list(col= tdr.IntervalDTH());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalDTM

Returns an R list representing the column definition for an instance of the INTERVAL\_DTM\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalDTM()
```



**Example: Retrieve a Column Definition For the INTERVAL\_DTM\_DT Type**

```
coldef <- list(col= tdr.IntervalDTM());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalDTS

Returns an R list representing the column definition for an instance of the INTERVAL\_DTS\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalDTS()
```

**Example: Retrieve a Column Definition For the INTERVAL\_DTS\_DT Type**

```
coldef <- list(col= tdr.IntervalDTS());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalHour

Returns an R list representing the column definition for an instance of the INTERVAL\_HOUR\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalHour()
```

**Example: Retrieve a Column Definition For the INTERVAL\_HOUR\_DT Type**

```
coldef <- list(col= tdr.IntervalHour());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalHTM

Returns an R list representing the column definition for an instance of the INTERVAL\_HTM\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalHTM()
```

**Example: Retrieve a Column Definition For the INTERVAL\_HTM\_DT Type**

```
coldef <- list(col= tdr.IntervalHTM());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalHTS

Returns an R list representing the column definition for an instance of the INTERVAL\_HTS\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalHTS()
```

**Example: Retrieve a Column Definition For the INTERVAL\_HTS\_DT Type**

```
coldef <- list(col= tdr.IntervalHTS());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalMinute

Returns an R list representing the column definition for an instance of the INTERVAL\_MINUTE\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalMinute()
```

**Example: Retrieve a Column Definition For the INTERVAL\_MINUTE\_DT Type**

```
coldef <- list(col= tdr.IntervalMinute());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalMonth

Returns an R list representing the column definition for an instance of the INTERVAL\_MONTH\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalMonth()
```

**Example: Retrieve a Column Definition For the INTERVAL\_MONTH\_DT Type**

```
coldef <- list(col= tdr.IntervalMonth());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalMTS

Returns an R list representing the column definition for an instance of the INTERVAL\_MTS\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalMTS()
```

**Example: Retrieve a Column Definition For the INTERVAL\_MTS\_DT Type**

```
coldef <- list(col= tdr.IntervalMTS());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalSecond

Returns an R list representing the column definition for an instance of the INTERVAL\_SECOND\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalSecond()
```

**Example: Retrieve a Column Definition For the INTERVAL\_SECOND\_DT Type**

```
coldef <- list(col= tdr.IntervalSecond());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalYear

Returns an R list representing the column definition for an instance of the INTERVAL\_YEAR\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalYear()
```

**Example: Retrieve a Column Definition For the INTERVAL\_YEAR\_DT Type**

```
coldef <- list(col= tdr.IntervalYear());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IntervalYTM

Returns an R list representing the column definition for an instance of the INTERVAL\_YTM\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

**Syntax**

```
tdr.IntervalYTM()
```

**Example: Retrieve a Column Definition For the INTERVAL\_YTM\_DT Type**

```
coldef <- list(col= tdr.IntervalYTM());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.IsDimension

Returns an integer indicating whether the input stream is a DIMENSION stream—1 for yes, 0 for no.

**Syntax**

```
tdr.IsDimension(streamno)
```

**Syntax Elements*****streamno***

Parameter type: integer

The input stream number.

**Example: Check If a Stream is a DIMENSION Input Stream**

```
stream <- 2;
X <- tdr.IsDimension(stream);
```

X is of type integer, and the value of X can be 0 or 1.

## tdr.LobAppend

Appends raw data to the end of the output LOB associated with a specified output locator. Returns a status code.

### Syntax

```
tdr.LobAppend( locator, data)
```

### Syntax Elements

#### *locator*

Parameter type: integer

The output locator returned by the tdr.LobCol2Loc function.

#### *data*

Parameter type: raw

A vector with raw data to be appended to the output LOB.

### Usage Notes

Before you call this function, you must call the tdr.LobCol2Loc function to get the output locator for the LOB where the data will be appended.

#### Note:

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Store an R Data Structure as a LOB in a Table

This example shows how an R data structure can be stored as a LOB in a table with two attributes: (int, blob):

1. Call the glm R function to compute a model and return an R data structure.
2. Serialize the data structure.
3. Store the data structure in the second column of the current output row.

```
lin_model <- glm(cc_rev ~ income + age + avg_ck_bal + avg_sv_bal, data
= training)
slin_model <- serialize( lin_model, NULL )
loc <- tdr.LobCol2Loc(0,1)
return_value <- tdr.LobAppend (loc, slin_model)
```

## tdr.LobClose

Closes an input LOB.

### Syntax

```
tdr.LobClose(contextID)
```

### Syntax Elements

#### *contextID*

Parameter type: integer

The context ID returned by the tdr.LobOpen\_CL function.

### Usage Notes

Before you call this function, you must call the tdr.LobOpen\_CL function to get the context ID. Then pass this context ID as an argument to this function.

#### Note:

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Close an Input LOB

Consider an input table with 5 attributes: (int, int, int, int, clob). This example shows how to read the last attribute of the current row as follows:

- Get the LOB locator
- Open the LOB
- Read the LOB
- Close the LOB

```
# Open the input stream
inHandle <- tdr.Open("R", 0, 0);

# Get the 5th attribute in the current row (LOB locator)
att5 <- tdr.GetAttributeByNdx(inHandle, 4, NULL);

# Open the LOB for reading
inlob <- tdr.LobOpen_CL(att5,0,0);

# Read the LOB, convert it to character, and print it
```

```
string <- tdr.LobRead(inlob$contextID, inlob$LOBlen);
print(rawToChar(string$buffer));

# Close the LOB
tdr.LobClose(inlob$contextID);
```

## tdr.LobCol2Loc

Computes the output LOB locator needed to append data to a LOB in the current output row. Returns an integer representing the output LOB locator.

### Syntax

```
tdr.LobCol2Loc( streamno, attributenr)
```

### Syntax Elements

#### *streamno*

Parameter type: integer

The output stream number.

#### *attributenr*

Parameter type: integer

The attribute number, starting from 0.

### Usage Notes

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Get the Output LOB Locator

Consider an output table with 5 attributes: (int, int, int, int, clob).

The following statement gets the output locator of the last attribute in the current row:

```
loc <- tdr.LobCol2Loc(0,4)
```

## tdr.LobOpen\_CL

Opens a LOB for reading. Returns a list to use to read the LOB. The list has two elements, contextID and LOBlen.

## Syntax

```
tdr.LobOpen_CL(locator, start, maxlength)
```

## Syntax Elements

### *locator*

Parameter type: integer

The locator of the LOB stored in the table.

### *start*

Parameter type: numeric

Start reading at this position in the LOB.

### *maxlength*

Parameter type: numeric

The maximum number of bytes to read.

## Usage Notes

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Open a LOB for Reading

Consider an input table with 5 attributes: (int, int, int, int, clob). This example shows how to read the last attribute of the current row as follows:

- Get the LOB locator
- Open the LOB
- Read the LOB
- Close the LOB

```
# Open the input stream
inHandle <- tdr.Open("R", 0, 0);

# Get the 5th attribute in the current row (LOB locator)
att5 <- tdr.GetAttributeByNdx(inHandle, 4, NULL);

# Open the LOB for reading
inlob <- tdr.LobOpen_CL(att5,0,0);
```



```
# Read the LOB, convert it to character, and print it
string <- tdr.LobRead(inlob$contextID, inlob$LOBlen);
print(rawToChar(string$buffer));

# Close the LOB
tdr.LobClose(inlob$contextID);
```

## tdr.LobRead

Reads a LOB. Returns a raw vector with the LOB content.

If an error occurs, the function returns NULL.

### Syntax

```
tdr.LobRead(contextID, length)
```

### Syntax Elements

#### *contextID*

Parameter type: integer

The context ID returned by the tdr.LobOpen\_CL function.

#### *length*

Parameter type: numeric

The number of bytes to read.

### Usage Notes

Before you call this function, you must call the tdr.LobOpen\_CL function to get the context ID. Then pass this context ID as an argument to this function.

#### Note:

This function is valid only if called from the table operator.

An error is raised if the function is called from the contract function.

### Example: Read a LOB

Consider an input table with 5 attributes: (int, int, int, int, clob). This example shows how to read the last attribute of the current row as follows:

- Get the LOB locator

- Open the LOB
- Read the LOB
- Close the LOB

```
# Open the input stream
inHandle <- tdr.Open("R", 0, 0);

# Get the 5th attribute in the current row (LOB locator)
att5 <- tdr.GetAttributeByNdx(inHandle, 4, NULL);

# Open the LOB for reading
inlob <- tdr.LobOpen_CL(att5,0,0);

# Read the LOB, convert it to character, and print it
string <- tdr.LobRead(inlob$contextID, inlob$LOBlen);
print(rawToChar(string$buffer));

# Close the LOB
tdr.LobClose(inlob$contextID);
```

## tdr.Open

Opens an input or output stream. Returns a handle of type raw to the iterator associated with the stream. You can use the handle to read or write rows and to close the iterator.

If an error occurs, the function returns NULL.

### Syntax

```
tdr.Open( direction, streamno, options )
```

### Syntax Elements

#### *direction*

Parameter type: character

Valid values are as follows:

- "R" (input)
- "W" (output)

#### *streamno*

Parameter type: integer

The stream number.

### ***options***

Parameter type: integer

How the stream is manipulated.

---

#### **Note:**

The only valid value for *options* is 0, which indicates that individual attributes can be accessed or modified.

---

## **Usage Notes**

The function opens an input or output stream. Once the stream is opened, rows can be read or written.

You can open an input stream multiple times within a table operator invocation. Each open resets the read position to the start of the stream. If you specify the PARTITION BY clause, opening a stream refers to the rows within a partition. If you do not specify a PARTITION BY clause, opening a stream refers to the rows within the AMP.

You can open an output stream a single time within a table operator invocation.

---

#### **Note:**

This function is valid only if called from the table operator.

---

The following are not allowed and result in an error:

- Open an input or output stream that is already open.
- Close and reopen an output stream.
- Call this function from the contract function.

### **Example: Opening an Input Stream**

# Open the input stream.

```
library(tdr);
stream <- 0;
options <- 0;
direction <- "R";
inHandle <- tdr.Open(direction, stream, options);
```

## **tdr.Read**

Reads a row from an input stream. Returns one of the following codes:

Return Code	Description
0	The operation was successful.
-1	The end of stream was reached.
-2	The table operator aborted.
-3	The input parameter has an invalid type.
-4	The input parameter has an invalid value.
-5	This function was called from the contract function.

## Syntax

```
tdr.Read( handle )
```

## Syntax Elements

### *handle*

Parameter type: raw vector

The handle of the input stream returned by the tdr.Open function.

## Usage Notes

Before you call this function, you must call the tdr.Open function to open the input stream. Then pass the handle returned from tdr.Open as an argument to this function.

### Note:

This function is valid only if called from the table operator.

## Example: Reading Rows From the Input Stream

This example opens an input stream and counts the number of rows successfully read from the input stream.

# Open the input stream.

```
library(tdr);
stream <- 0;
options <- 0;
direction <- "R";
inHandle <- tdr.Open(direction, stream, options);
```

# Read rows from the input stream.

```
nrrows <- 0;
while (tdr.Read(inHandle) == 0) {
  nrrows <- nrrows+1;
}
```

## tdr.Real

Returns an R list representing the column definition for an instance of the REAL\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Real()
```

### Example: tdr.Real - Retrieve a Column Definition For the REAL\_DT Type

```
coldef <- list(col= tdr.Real());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.SetAttributeByNdx

Sets the value of an individual attribute in the current output stream row.

### Syntax

```
tdr.SetAttributeByNdx( handle, index, attribute, coldef)
```

### Syntax Elements

#### *handle*

Parameter type: raw vector

The handle of the output stream returned by the tdr.Open function.

#### *index*

Parameter type: integer

The index of an attribute. The valid range is from 0 to  $n-1$ , where  $n$  is the number of attributes in the stream. Index 0 indicates the first attribute.

#### *attribute*

Parameter type: list

The attribute value to be set in the form of a list containing the value and a null indicator. This is the return value of a call to the `tdr.GetAttributeByNdx` function.

### ***coldef***

Parameter type: list

The schema of a stream. This schema is a list with the number of columns and the definition of each column. Each definition includes the column information for the data type of that specific column. This is the return value of a call to the `tdr.GetColDef` function.

If *coldef* is NULL, the function will retrieve the *coldef* information.

## **Usage Notes**

The *coldef* parameter can be used to improve performance. By passing the *coldef* information to the function, the function does not have to retrieve it on each call. However, you can optionally pass NULL instead and the function will retrieve the *coldef* information by itself.

Before you call this function, you must call the `tdr.Open` function to open the output stream. Then pass the handle returned from `tdr.Open` as an argument to this function.

---

### **Note:**

This function is valid only if called from the table operator.

---

The function raises an error in the following situations:

- The handle is associated with an input stream.
- The output stream is not open.
- The index is not valid.
- The *coldef* list contains an invalid data type or an incomplete set of attributes for that data type.
- The attribute is not of type list.
- The type of the input attribute is different from the type of attribute at the given index.
- This function was called from the contract function.

### **Example: Set the First Attribute in the Current Row of the Output Stream**

This example sets the first attribute in the current row of the output stream to value 123.

```
myatt <- list ( value=as.integer(123), nullIndicator=0 );
tdr.SetAttributeByNdx(outHandle, 0, myatt, NULL);
```

In the previous example, NULL was passed as the *coldef* parameter; therefore, the function retrieves the *coldef* information.

In the following example, the *coldef* information is passed to the function. This would improve performance.

```
myatt <- list ( value=as.integer(123),nullIndicator=0);
coldef <- tdr.GetColDef(streamno, direction);
tdr.SetAttributeByNdx(outHandle, 0, myatt, coldef);
```

## tdr.SetContractDef

Sets the contract context.

### Syntax

```
tdr.SetContractDef( ctx )
```

### Syntax Elements

#### *ctx*

Parameter type: raw

An opaque sequence of bytes that can be retrieved by the table operator.

### Usage Notes

This function is valid only if called from the contract function.

An error is raised if the function is called from the table operator.

### Example: Set the Contract Context

The following statements illustrate how this function can be used in the contract function:

1. A data structure is created.
2. The structure is serialized.
3. The serialized version of the structure is set as the contract context.

```
myctx <- list( division="appliances", codes=c(10, 34) );
myctxS <- serialize( myctx, NULL );
tdr.SetContractDef( myctxS );
```

## tdr.SetError

Sets an error code and message to be displayed on the console when the query invoking the table operator is executed and an error is encountered.

### Syntax

```
tdr.SetError(code, message)
```

## Syntax Elements

### *code*

Parameter type: character

An error code.

### *message*

Parameter type: character

An error message.

## Example: Displaying an Error Code and Message

Suppose the data in the input table has invalid values. You can report the error using the following statement which prints out the error code "U1003" with message "Invalid Value for Division Number".

```
tdr.SetError("U1003", "Invalid Value for Division Number")
```

## tdr.SetFormat

Sets an attribute of the format for a specified stream.

## Syntax

```
tdr.SetFormat( attribute, streamno, direction, value )
```

## Syntax Elements

### *attribute*

Parameter type: character

The format attribute to be set.

Currently, the only valid value for *attribute* is "RECFMT", which defines the input/output record format.

### *streamno*

Parameter type: integer

The stream number.

### *direction*

Parameter type: character



Valid values are as follows:

- "R" (input)
- "W" (output)

### ***value***

Parameter type: integer

The new value for the format attribute.

The valid values are as follows:

- 1 for IndicData with row separator sentinels
- 2 for IndicData with no row or partition separator sentinels

## **Usage Notes**

This function is valid only if called from the contract function.

An error is raised if this function is called from the table operator.

### **Example: Set the Record Format of an Input Stream**

The following statement sets the record format of input stream 0 to IndicData with row separator sentinels:

```
tdr.SetFormat("RECFMT", 0, "R", 1);
```

## **tdr.SetHashByDef**

Sets the HASH BY specification for a specified input stream.

### **Syntax**

```
tdr.SetHashByDef( streamno, hashbydef )
```

### **Syntax Elements**

#### ***streamno***

Parameter type: integer

The input stream number.

#### ***hashbydef***

Parameter type: list

A vector of column names.

## Usage Notes

This function is valid only if called from the contract function.

The function must run on a PE-only node.

An error is raised in the following situations:

- The function is called from the table operator.
- The HASH BY metadata was already set.

## Example: Set the HASH BY Specification for an Input Stream

The following statement sets the HASH BY metadata for input stream 0 with the specification that the columns in the HASH BY clause are col1 and col3.

```
hashbydef <- c("col1", "col3");
tdr.SetHashByDef( 0, hashbydef );
```

## tdr.SetLocalOrderByDef

Sets the ordering specification for a specified input stream.

### Syntax

```
tdr.SetLocalOrderByDef( streamno, lorderbydef )
```

### Syntax Elements

#### *streamno*

Parameter type: integer

The input stream number.

#### *lorderbydef*

Parameter type: list

A list of vectors with the following:

- Column names
- Ordering of the columns
- NULLS FIRST/NULLS LAST indicators

## Usage Notes

This function is valid only if called from the contract function.

The function must run on a PE-only node.

An error is raised in the following situations:

- The function is called from the table operator.
- The LOCAL ORDER BY metadata was already set.

### Example: Set the LOCAL ORDER BY Specification for an Input Stream

The following statement sets the LOCAL ORDER BY specification for input stream 0 to be col1 ASC NULLS FIRST, col3 DESC NULLS LAST.

```
lorderbydef <- list(c("col1","A","NF"),c("col2","D","NL"))
tdr.SetLocalOrderByDef(0, lorderbydef );
```

## tdr.SetOutputColDef

Sets the schema of the output stream.

### Syntax

```
tdr.SetOutputColDef( streamno, coldef )
```

### Syntax Elements

#### *streamno*

Parameter type: integer

The output stream number.

#### *coldef*

Parameter type: list

The schema of a stream. This schema is a list with the number of columns and the definition of each column. Each definition includes the following column information:

- Type  
Valid data type values are defined by the dtype\_en enumeration in the sqltypes\_td.h header file.
- Size (in bytes)
- Character type for textual fields:
  - 1 for LATIN
  - 2 for UNICODE

### Usage Notes

This function is valid only if called from the contract function.

An error is raised if the function is called from the table operator.

### Example: Set the Column Definitions of an Output Stream

This example sets the column definitions of an output stream with three attributes (COL1 int, COL2 real, COL3 varchar(30)).

```
stream <- 0;
integer = list( datatype="INTEGER_DT", bytesize=4 );
real <- list( datatype="REAL_DT", bytesize=8 );
varchar30 <- list(datatype="VARCHAR_DT", charset="LATIN_CT", size.length=30);
coldef <- list(COL1=integer, COL2=real, COL3=varchar30);
tdr.SetOutputColDef(stream, coldef);
```

## tdr.SmallInt

Returns an R list representing the column definition for an instance of the SMALLINT\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.SmallInt()
```

### Example: Retrieve a Column Definition For the SMALLINT\_DT Type

```
coldef <- list(col= tdr.SmallInt());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.TblRead

Reads an input stream into an R data frame and returns the data frame.

If there are no more rows in the stream, the function returns an empty data frame.

If an error occurs, the function returns one of the following error codes:

Error Code	Error Condition
-2	The table operator aborted.
-3	The input parameter has an invalid type.
-4	The input parameter has an invalid value.
-5	This function was called from the contract function.

## Syntax

```
tdr.TblRead( handle, dataframesize )
```

## Syntax Elements

### *handle*

Parameter type: raw vector

The handle of the input stream returned by the tdr.Open function.

### *dataframesize*

Parameter type: integer

The maximum size (in bytes) of the data frame that is generated. This size is limited by the UDFmalloc size of the secure server which is 32MB by default.

If you set this parameter to 0, the available size is divided equally between the input and output streams.

## Usage Notes

Before you call this function, you must call the tdr.Open function to open the input stream. Then pass the handle returned from tdr.Open as an argument to this function.

### Note:

Based on the memory resource available, the data frame size returned may be less than the size specified in the call to tdr.TblRead.

This function is valid only if called from the table operator.

The following data types are not supported in data frames:

- DECIMAL8
- DECIMAL16
- TIME
- TIME\_WTZ
- TIMESTAMP
- TIMESTAMP\_WTZ
- INTERVAL\_YTM
- INTERVALH
- INTERVALM
- INTERVALS
- INTERVAL\_HTM

- INTERVAL\_HTS
- INTERVAL\_MTS
- INTERVAL\_SECOND

If you want to use these data types in data frames, you must cast them to a supported type.

The data frame has the maximum number of rows that can fit in allocated memory. The data frame is created by mapping Teradata data types to R data types. See [Mapping Between Teradata Data Types and R Types](#).

## tdr.TblWrite

Writes an R data frame to the output stream. Returns one of the following codes:

Return Code	Description
0	The operation was successful.
-1	The end of stream was reached.
-2	The table operator aborted.
-3	The input parameter has an invalid type.
-4	The input parameter has an invalid value.
-5	This function was called from the contract function.

### Syntax

```
tdr.TblWrite( handle, df )
```

### Syntax Elements

#### *handle*

Parameter type: raw vector

The handle of the output stream returned by the tdr.Open function.

#### *df*

Parameter type: data frame

An R data frame containing the rows to be written.

### Usage Notes

Before you call this function, you must call the tdr.Open function to open the output stream. Then pass the handle returned from tdr.Open as an argument to this function.

**Note:**

This function is valid only if called from the table operator.

The following data types are not supported in data frames:

- DECIMAL8
- DECIMAL16
- TIME
- TIME\_WTZ
- TIMESTAMP
- TIMESTAMP\_WTZ
- INTERVAL\_YTM
- INTERVALH
- INTERVALM
- INTERVALS
- INTERVAL\_HTM
- INTERVAL\_HTS
- INTERVAL\_MTS
- INTERVAL\_SECOND

If you want to use these data types in data frames, you must cast them to a supported type.

## tdr.Time

Returns an R list representing the column definition for an instance of the TIME\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.Time(precision)
```

### Syntax Elements

#### *precision*

Parameter type: integer (range 0 to 6 inclusive)

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

### Example: Retrieve a Column Definition For the TIME\_DT Type

```
coldef <- list(col= tdr.Time(6));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.Timestamp

Returns an R list representing the column definition for an instance of the `TIMESTAMP_DT` data type with the given attributes, to use with the `tdr.SetOutputColDef` function.

### Syntax

```
tdr.Timestamp(precision)
```

### Syntax Elements

#### *precision*

Parameter type: integer (range 0 to 6 inclusive)

A single digit representing the number of significant digits in the fractional portion of the `SECOND` field.

### Example: Retrieve a Column Definition For the `TIMESTAMP_DT` Type

```
coldef <- list(col= tdr.Timestamp(6));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.TimestampWTZ

Returns an R list representing the column definition for an instance of the `TIMESTAMP_WTZ_DT` data type with the given attributes, to use with the `tdr.SetOutputColDef` function.

### Syntax

```
tdr.TimestampWTZ()
```

### Example: Retrieve a Column Definition For the `TIMESTAMP_WTZ_DT` Type

```
coldef <- list(col= tdr.TimestampWTZ());
tdr.SetOutputColDef(stream, coldef);
```

## tdr.TimeWTZ

Returns an R list representing the column definition for an instance of the `TIME_WTZ_DT` data type with the given attributes, to use with the `tdr.SetOutputColDef` function.



**Syntax**

```
tdr.TimeWTZ(precision)
```

**Syntax Elements*****precision***

Parameter type: integer (range 0 to 6 inclusive)

A single digit representing the number of significant digits in the fractional portion of the SECOND field.

**Example: Retrieve a Column Definition For the TIME\_WTZ\_DT Type**

```
coldef <- list(col= tdr.TimeWTZ(6));
tdr.SetOutputColDef(stream, coldef);
```

**tdr.tracestring**

Returns the current setting of the trace string that the SET SESSION FUNCTION TRACE statement set up, a character vector.

**Syntax**

```
tdr.tracestring()
```

**Related Information**

For more information on SET SESSION FUNCTION TRACE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

**tdr.tracewrite**

Writes trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement.

**Syntax**

```
tdr.tracewrite(values)
```

**Syntax Elements*****values***

Parameter type: list

A list of values to write to the columns in a temporary trace table.

## Related Information

For more information on CREATE GLOBAL TEMPORARY TRACE TABLE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## tdr.VarByte

Returns an R list representing the column definition for an instance of the VARBYTE\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.VarByte(size)
```

### Syntax Elements

#### size

Parameter type: integer

The number of bytes allotted to this column.

### Example: Retrieve a Column Definition For the VARBYTE\_DT Type

```
coldef <- list(col= tdr.VarByte(100));
tdr.SetOutputColDef(stream, coldef);
```

## tdr.VarChar

Returns an R list representing the column definition for an instance of the VARCHAR\_DT data type with the given attributes, to use with the tdr.SetOutputColDef function.

### Syntax

```
tdr.VarChar(size, chartype)
```

### Syntax Elements

#### size

Parameter type: integer

The number of characters or bytes allotted to this column.

**chartype**

Parameter type: string

The character set for the VARCHAR column being defined.

**Example: Retrieve a Column Definition For the VARCHAR\_DT Type**

```
coldef <- list(col= tdr.VarChar(100,"LATIN_CT"));
tdr.SetOutputColDef(stream, coldef);
```

**tdr.Write**

Writes a row to the output stream. Returns one of the following codes:

Return Code	Description
0	The operation was successful.
-1	The end of stream was reached.
-2	The table operator aborted.
-3	The input parameter has an invalid type.
-4	The input parameter has an invalid value.
-5	This function was called from the contract function.

**Syntax**

```
tdr.Write( handle )
```

**Syntax Elements****handle**

Parameter type: raw vector

The handle of the output stream returned by the tdr.Open function.

**Usage Notes**

Before you call this function, you must call the tdr.Open function to open the output stream. Then pass the handle returned from tdr.Open as an argument to this function.

**Note:**

This function is valid only if called from the table operator.

**Example: Write the Current Row to the Spool**

The following statement writes the current row to the spool. Before calling `tdr.Write`, you must set the values of individual attributes in the current row using the `tdr.SetAttributeByNdx` function.

```
tdr.Write( outHandle );
```

# Global and Persistent Data

The following sections provide conceptual, reference, and procedural topics that are specific to using global and persistent (GLOP) data in C and C++ UDFs, UDMs, and external stored procedures (collectively called *external routines*).

Before you can use GLOP data, you must execute the DIPGLOP script to create the DBCExtension database and the tables and stored procedures used to manage and manipulate GLOP data. For information on executing the DIPGLOP script using the Database Initialization Program (DIP) utility, see *Teradata Vantage™ - Database Utilities*, B035-1102.

Type of Information	Topics
Concepts	<ul style="list-style-type: none"> <li>• <a href="#">About GLOP Data</a></li> <li>• <a href="#">GLOP Mappings</a></li> <li>• <a href="#">GLOP Set</a></li> </ul>
Procedures	<ul style="list-style-type: none"> <li>• <a href="#">Managing GLOP Data</a></li> <li>• <a href="#">Creating a GLOP Set Definition</a></li> <li>• <a href="#">Adding Mappings and Data to a GLOP Set</a></li> <li>• <a href="#">Becoming a Member of a GLOP Set</a></li> <li>• <a href="#">Using GLOP Data in a C/C++ External Routine</a></li> </ul>
Reference	<ul style="list-style-type: none"> <li>• <a href="#">DBCExtension.GLOP_Add Stored Procedure</a></li> <li>• <a href="#">DBCExtension.GLOP_Remove Stored Procedure</a></li> <li>• <a href="#">DBCExtension.GLOP_Change Stored Procedure</a></li> <li>• <a href="#">DBCExtension.GLOP_Report Stored Procedure</a></li> <li>• <a href="#">System Disk Space</a></li> <li>• <a href="#">Global Configuration Settings</a></li> </ul>

For details on the C library functions that you can use to access GLOP data in a C or C++ external routine, see [C Library Functions](#).

## About GLOP Data

GLOP data is a type of memory mapped data available to external routines where the data persists beyond the life of one invocation of an external routine.

The persistence, as well as the data that is presented to a specific external routine, is based on specific boundaries such as a role or user account.

For example, consider GLOP data that is based on a user account. The data that an external routine has access to is based on the user who invokes the external routine. The data is available for an external routine to use until the user logs off.

Using stored procedures in the DBCEXTENSION system database, you place GLOP data (as a BLOB) in a system table (also in the DBCEXTENSION database) with particular specifications. The data from this table is available in memory on all nodes and is presented to external routines that are properly configured to use the data.

### Related Information

- [GLOP Types](#)
- [Persistence of GLOP Types](#)
- [GLOP Data Attributes](#)

## Benefits of GLOP Data

You can use GLOP data to solve a number of operational issues.

Here are some examples:

- Consider an external routine that requires access to specific operational data. If the volume of data is large or if the data must be secure, passing the data as an argument is not practical. You could place the data in external files, but the cost of maintenance and computing resources is high, especially for a multiple node system or if simultaneous external routine invocations use the same data.

Instead, you can use system stored procedures to manage GLOP data. The data is available on all nodes and is available to external routines that you configure to use it.

- Suppose an external routine needs to access or change data differently depending on the user or role associated with the external routine invocation.

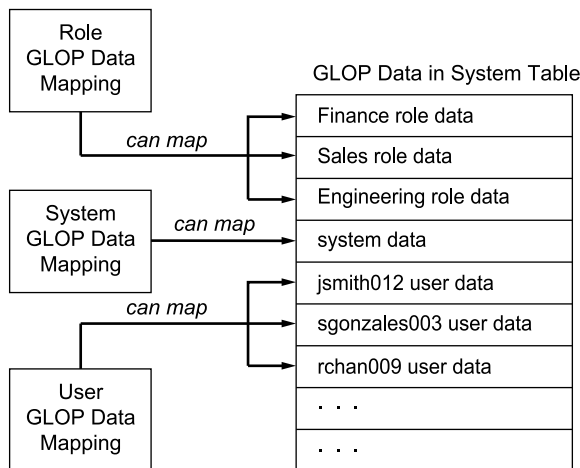
The external routine can use GLOP data that is associated with a user or role, and the actual data the system makes available to the external routine is different depending on the current role or user associated with an external routine invocation.

## GLOP Mappings

You define GLOP mappings to map specific types of GLOP data, for example role type data or user type data. The system uses the context in which an external routine runs to map the actual data that the external routine can access.

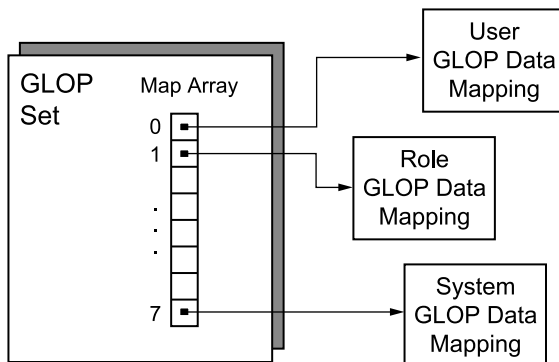
For example, you can define a GLOP mapping for role type data. The actual GLOP data that the system maps for a specific external routine invocation depends on the current role for the user that executes the external routine. If the user does not have a role, or if there is no GLOP data for the current role for the user, the external routine has no access to role type data.

The following diagram illustrates how three different GLOP mappings can potentially map to actual data that you store in the system table.



## GLOP Set

GLOP data is organized into what is called a GLOP set consisting of up to eight GLOP data mappings of various types.



Although a GLOP set can consist of up to eight GLOP data mappings, some GLOP types impose further restrictions as to the number of mappings of that type in a GLOP set.

GLOP Type	Maximum Number of Mappings in a GLOP Set
External Routine	1
Role	
User	
Request	8
Session	
System	

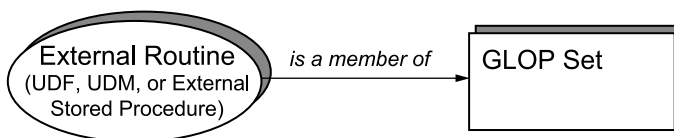
GLOP Type	Maximum Number of Mappings in a GLOP Set
Transaction	

### Related Information

- [GLOP Types](#)
- [Creating a GLOP Set Definition](#)
- [Adding Mappings and Data to a GLOP Set](#)

## GLOP Set Membership

When you create an external routine, for example when you use the CREATE FUNCTION statement to create a UDF, you can specify the USING GLOP SET option to make the external routine a member of a GLOP set.



An external routine can only be a member of one GLOP set, but any number of external routines can be members of the same GLOP set.

### Related Information

[Becoming a Member of a GLOP Set](#)

## GLOP Data Access From an External Routine

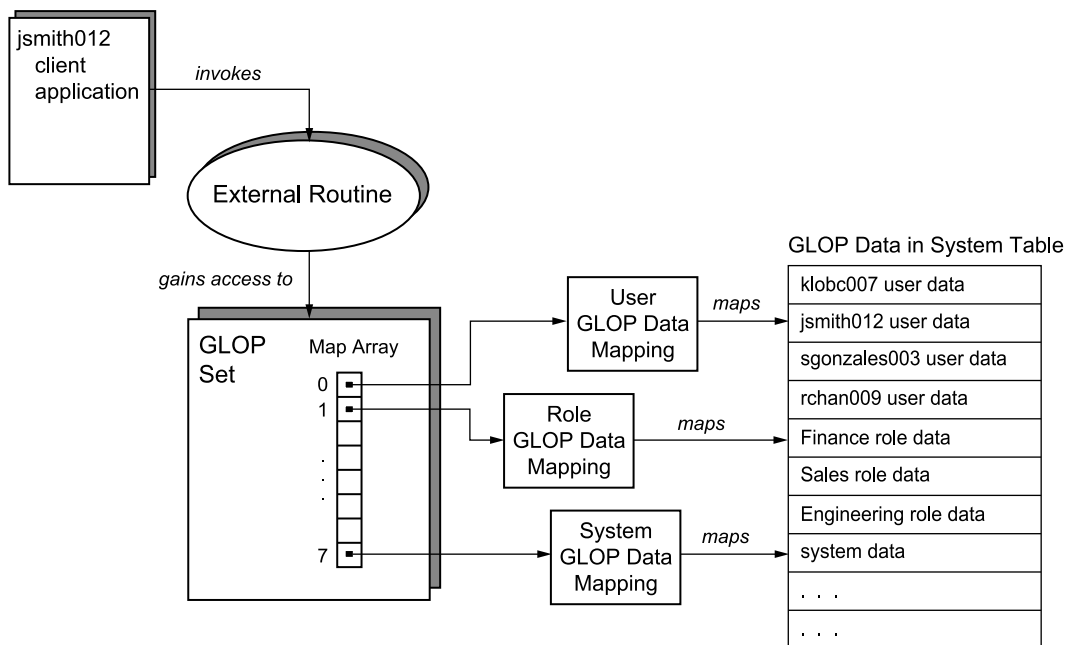
To access GLOP data, a C or C++ external routine must first use the FNC\_Get\_GLOP\_Map library function to gain access to the GLOP set of which the external routine is a member.

The function returns a pointer to a structure that is an array of eight data references, the maximum number of GLOP mappings in a GLOP set.

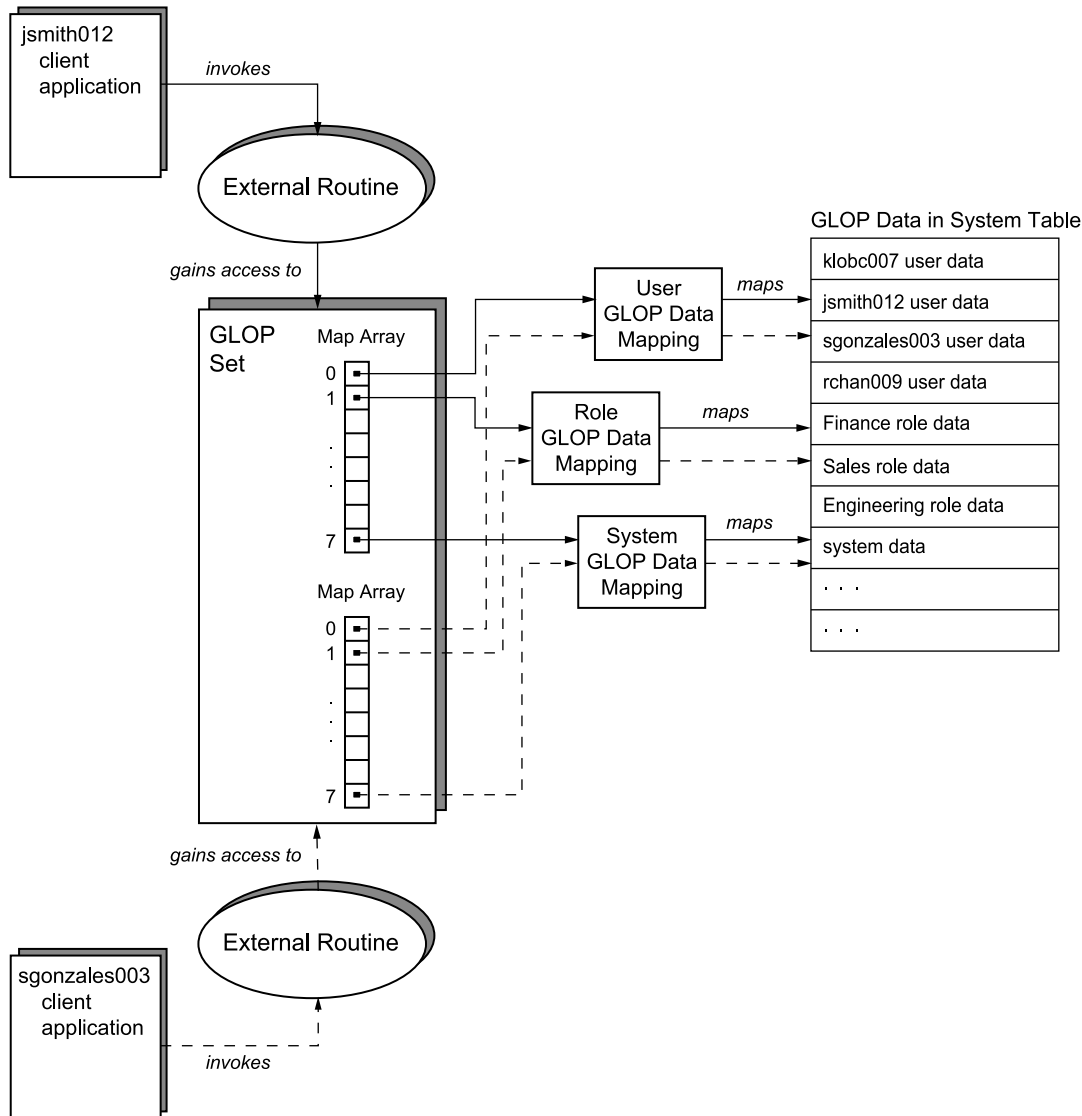
The context in which a particular external routine runs determines the actual data that the external routine has access to.

For example, the following diagram illustrates the GLOP data available to an external routine that is invoked by user jsmith012 who has been granted a Finance role.





If the external routine is also invoked by user sgonzales003, who has been granted the Sales role, the GLOP data that is available can be illustrated like this:

**Related Information:**

[Using GLOP Data in a C/C++ External Routine](#)

## GLOP Page

You can split read-only GLOP data into a number of pages, each of which can be overlaid with the current GLOP data in memory. The page to initially map for read-only GLOP data can be specified when you call `DBCExtension.GLOP_Add` to add the GLOP data or `DBCExtension.GLOP_Change` to change the GLOP data.

Different external routines can map different pages for read-only GLOP data at the same time.

Read/write or globally modifiable GLOP data can only consist of one page.

## Related Information

For details on the read-only, read/write, and globally modifiable modification attributes, see [GLOP Modification Attributes](#).

## GLOP Types

The GLOP type determines the persistence boundary of the GLOP data, based on particular database activity.

### System GLOP

A system GLOP mapping is mapped by all members of a given set. All external routines that are members of the same named GLOP set share this memory mapping.

The modification attribute (read-only or read/write) that you set for system GLOP data has the following effect.

IF the modification attribute is ...	THEN ...
read-only	every external routine that is a member of the set is mapped with the same data. They all see the same context. If the system GLOP consists of multiple pages, then different external routines can simultaneously map different pages of the system GLOP, enabling each routine with a way to see a part of the data that it is interested in.
read/write	every external routine that is a member of the set on that one vproc sees the same changes made by any external routine that changes the data. Different vprocs do not necessarily have the same data changes unless the external routine that changes the data makes the same changes on all vprocs.

For more information on the read-only and read/write modification attributes, see [GLOP Modification Attributes](#).

For system GLOP data with a read/write modification attribute, the mapping attribute (normal or shared) that you set has the following effect.

IF the mapping attribute is ...	THEN ...
normal	different GLOP sets that map the same system data get separate mappings.
shared	different GLOP sets that map the same system data get the same system GLOP mapping on a vproc.

For more information on the normal and shared mapping attributes, see [GLOP Mapping Attributes](#).

## Role GLOP

A role GLOP mapping is always associated with a particular role that the external routine is executing and is based on what was set up for the logged on user for a particular session.

IF ...	THEN ...
a mapping exists for a given role and the session has its context set up to use the role	the system maps the GLOP data for the given role.
the external routine runs in a context that does not have a role GLOP data mapping specified for the given role in the GLOP set	nothing is mapped for the external routine.
the routine is set to run in a session that has role ALL set up	nothing is mapped, because it is not possible to determine which role to use.
the session role is NONE or NULL	nothing is mapped.

Role GLOP data mapping for a GLOP set is not session local, it is based on the role. So, if there are ten sessions and each of them uses the same role then those ten sessions share the same role GLOP data mapping in all external routines that are members of the same GLOP set. This is similar to the system GLOP mapping except that there is a separate mapping for each role in the GLOP set.

If other external routines are members of a different GLOP set and the same role is specified in both sets, those role mappings are independent. (This assumes that there is actually GLOP data that identifies a particular role as having a mapping for a particular GLOP set.)

The effect of the modification attributes of a role GLOP is similar to the effect of the modification attributes of a system GLOP.

IF the modification attribute is ...	THEN ...
read-only	all vprocs have identical role data for each role that has a GLOP definition. In addition, there is only one physical copy of read-only role GLOP data on a node.
read/write	then the data can differ on different vprocs.

If you want several roles to map to the same data, then use the shared mapping attribute for those roles that map the same data and have them reference the same GLOP data.

For more information on the read-only and read/write modification attributes, see [GLOP Modification Attributes](#). For more information on the normal and shared mapping attributes, see [GLOP Mapping Attributes](#).

## User GLOP

A user GLOP mapping is always associated with a particular logged on user ID. It is not associated with a proxy user but the underlying user for the proxy. All sessions that are logged onto the same user ID share the data for the specific user mapping provided they specified the same GLOP set. This works similarly to

the role mapping. (For details, see [Role GLOP](#).) This allows several sessions all logged onto the same user to share information inside of the external routines that are associated with that user GLOP set.

The mapping is always associated with the logged on user that executes the external routine, not the current authorization (which could be different). For example, if Joe creates an external stored procedure and Betty logs on and executes that external 'DEFINER' procedure, the current authorization is Joe. But, Betty is the logged on user and if there is GLOP data for user Betty, that is the one that gets mapped, not the GLOP data for user Joe.

## Session GLOP

A session GLOP mapping is local to a particular session. Only that session can normally see the data. The data exists as long as the session is logged on. The data never survives past the logged on session. When a new session is started a new fresh copy from the specific GLOP data row for the set is obtained and mapped. The data is accessible to all external routines that execute within the session and the given GLOP set.

External routines that are members of different GLOP sets get different GLOP session mappings. Mappings are private to the session unless the shared mapping attribute is set.

IF the mapping attribute is ...	THEN ...
normal	different GLOP sets that map the same session read/write data get separate mappings.
shared	different GLOP sets that map the same session read/write data get the same mappings.

GLOP changes made to read/write data are retained and visible for the next external routine using the same GLOP set.

For more information on the normal and shared mapping attributes, see [GLOP Mapping Attributes](#).

## Transaction GLOP

A transaction GLOP mapping is local to a session and transaction. A transaction GLOP mapping survives as long as a transaction survives. This GLOP type is only available to external routines executing on an AMP vproc. It is not available or relevant to PE vprocs because many external routines actually execute outside of a transaction, so even if mapping were provided it would not be available a lot of the time.

When a transaction terminates, the mapping is lost.

When a transaction starts, a new fresh copy from the specific GLOP data for the set is obtained and mapped.

## Request GLOP

A request GLOP mapping is local to a session, transaction and request combination. This GLOP type is only available to external routines executing on an AMP vproc. A request GLOP mapping has potentially the shortest life. It is created when the mapping so specifies at the beginning of a request and survives for the duration of the request. All external routines executed within the same request and GLOP membership see

the same mapping for the session in question. At the end of the request the mapping is discarded. A new request gets a new mapping from the GLOP data associated with the GLOP membership.

Because requests are potentially very short running, using this GLOP type requires some very careful thought. If not careful, this mode could substantially increase the running time of a transaction, for example, a set of small requests encountered in TPUMP.

## External Routine GLOP

An external routine GLOP mapping is associated with a specific routine. It is always local to that routine, but it crosses all session boundaries. In other words, no matter in what session the external routine executes in, the mapping is always available. If there are several instances of the external routine with the same name running in different sessions they all look at the same GLOP data for this GLOP type. Changes made to it are seen by all future callers. The GLOP data is retained until the next system restart or a forced remove by a change of the GLOP.

## Persistence of GLOP Types

The GLOP type determines the persistence of a GLOP mapping.

IF the GLOP type is ...	THEN it persists until the ...
External Routine	system restarts.
Request	request completes or is rolled back.
Role	role is no longer in use.
Session	session logs off.
System	system is restarted.
Transaction	transaction ends or aborts.
User	user logs off.

For details on the meanings of the GLOP types, see [GLOP Types](#).

## GLOP Data Attributes

Certain attributes associated with GLOP data determine modification properties and how the GLOP set maps the GLOP data.

### GLOP Modification Attributes

GLOP modification attributes determine whether the GLOP is read-only, read/write, globally modifiable, or disabled.

Attribute	Description
Read-Only	Used for GLOP data that never changes. The memory access is actually set to read-only. Any attempt to change the GLOP data causes a system reset when running in unprotected mode or a segmentation violation error result when running in protected or secure mode.
Read/Write	Used for GLOP data that changes at run time. The memory can be read and written at any time. However, to prevent simultaneous updates from conflicting with each other for different external routines that might have the same GLOP data mapped, the best practice is to apply a write lock on the GLOP data by the writer and a read lock by the readers.
Globally Modifiable	Used for GLOP data that might be globally modified at run time. It is read/write GLOP data that can be changed on all vprocs on all nodes at once. If this mode is not set, the GLOP data cannot be updated in a global manner.
Disabled	Used for GLOP data that is not to be mapped. There is a definition for it in one or more sets but the access to it has been turned off.

You set the modification attributes for GLOP data when you call the `DBCExtension.GLOP_Add` stored procedure to add GLOP data and GLOP mappings to a GLOP set. For details, see [DBCExtension.GLOP\\_Add Stored Procedure](#).

## GLOP Mapping Attributes

GLOP mapping attributes determine how a GLOP set maps GLOP data.

Attribute	Description
Normal	Normal mapping mode means the GLOP data is mapped as indicated by the GLOP type specifications.
Shared	<p>Shared mapping mode overrides the normal GLOP data mapping and maps one shared copy of the GLOP data per vproc no matter what type it is. The mapping mode attribute is associated with a GLOP set. If several GLOP sets have the shared flag set for the same GLOP data, they get the same logical address to the GLOP data. If an additional set for the same GLOP data has the normal mode set, it gets a separate copy of GLOP data independent of the other shared one.</p> <p><b>Note:</b> This setting is not relevant for read-only GLOP data because only one copy is ever mapped per node.</p>

You set the mapping attributes for GLOP data when you call the `DBCExtension.GLOP_Add` stored procedure to add GLOP data and GLOP mappings to a GLOP set. For details, see [DBCExtension.GLOP\\_Add Stored Procedure](#).

## GLOP Data Ageing

Read-only GLOP data can be aged out of memory. The ageing timeout can be specified in the number of minutes of inactivity. This provides a way to reduce the use of memory and disk resources for unused GLOP mappings.

Read/write or globally modifiable GLOP data does not age out.

GLOP data that is session based is deleted when the session logs out, GLOP data that is transaction based is deleted when the transaction terminates, and GLOP data that is request based is deleted when the request ends.

## GLOP Data System Tables

The DBCEExtension database contains three system tables for managing GLOP data. The DBCEExtension database and the tables are created by using the Database Initialization Program (DIP) utility and executing the DIPGLOP script.

Table	Description
DBCEExtension.GLOP_Map	Contains possible GLOP mappings for a particular GLOP set.
DBCEExtension.GLOP_Set	Describes the GLOP set names and GLOP mapping types.
DBCEExtension.GLOP_Data	Describes the GLOP data to map for a particular GLOP mapping in a GLOP set.

You cannot access the tables directly. To access the tables, use the GLOP system stored procedures DBCEExtension.GLOP\_Add, DBCEExtension.GLOP\_Change, DBCEExtension.GLOP\_Remove, and DBCEExtension.GLOP\_Report.

For more information on the ...	See ...
GLOP data system tables	<i>Teradata Vantage™ - Database Administration</i> , B035-1093.
DIPGLOP script and DIP utility	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102.
GLOP data system stored procedures	<a href="#">DBCEExtension.GLOP_Add Stored Procedure.</a>
	<a href="#">DBCEExtension.GLOP_Remove Stored Procedure.</a>
	<a href="#">DBCEExtension.GLOP_Change Stored Procedure.</a>
	<a href="#">DBCEExtension.GLOP_Report Stored Procedure.</a>

## GLOP Data System Stored Procedures

The DBCEExtension database contains four stored procedures for manipulating GLOP data. The DBCEExtension database and the stored procedures are created by using the Database Initialization Program (DIP) utility and executing the DIPGLOP script.

Use the stored procedures to manipulate the GLOP data system tables in the DBCEExtension database.

Stored Procedure	Description
DBCEExtension.GLOP_Add	Add GLOP data and GLOP mappings to a GLOP set. For details, see <a href="#">DBCEExtension.GLOP_Add Stored Procedure.</a>



Stored Procedure	Description
DBCExtension.GLOP_Change	Change GLOP data and some of the settings. For details, see <a href="#">DBCExtension.GLOP_Change Stored Procedure</a> .
DBCExtension.GLOP_Remove	Delete a GLOP set or remove individual components from a GLOP set. For details, see <a href="#">DBCExtension.GLOP_Remove Stored Procedure</a> .
DBCExtension.GLOP_Report	Output specific information about a particular GLOP set or all GLOP sets. For details, see <a href="#">DBCExtension.GLOP_Report Stored Procedure</a> .

## Managing GLOP Data

Here is a list of the tasks involved in managing GLOP data:

1. Execute the DIPGLOP script using the Database Initialization Program (DIP) utility to create the DBCExtension database and its tables and stored procedures.  
For more information on DIP and the DIPGLOP script, see *Teradata Vantage™ - Database Utilities*, B035-1102.
2. Determine the type of GLOP data you need.  
For details on GLOP types, see [GLOP Types](#).
3. Create a GLOP set definition.  
For details, see [Creating a GLOP Set Definition](#).
4. Add the GLOP data and mappings to the GLOP set.  
For details, see [Adding Mappings and Data to a GLOP Set](#).
5. Write a C or C++ external routine that uses the GLOP data.  
For details, see [Using GLOP Data in a C/C++ External Routine](#).
6. Create a definition for the external routine that makes the external routine a member of the GLOP set.  
For details, see [Becoming a Member of a GLOP Set](#).

## Creating a GLOP Set Definition

To create a GLOP set definition, use the CREATE GLOP SET statement.

Here is an example that creates a GLOP set definition called GLOP\_Set\_1001:

```
CREATE GLOP SET GLOP_Set_1001;
```

For details on the CREATE GLOP SET statement and the authorization required to use it, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Adding Mappings and Data to a GLOP Set

You can add up to eight GLOP mappings to a GLOP set. To add a mapping and GLOP data to a GLOP set, use the `DBCExtension.GLOP_Add` system stored procedure. Because the data that you add to a GLOP set must have a BLOB data type, consider using a UDF that generates the data and returns it as a BLOB type.

Here is an example that adds a system type GLOP mapping and data to the `GLOP_Set_1001` GLOP set. A UDF called `GLOP_System_Data` generates the BLOB data.

```
CALL DBCExtension.GLOP_Add
('GLOP_Set_1001','SY', NULL, 'CompanyInfo',
 'N', 0, 'N', 'W', 'E', 0, -1, 1, GLOP_System_Data());
```

For details on the `DBCExtension.GLOP_Add` system stored procedure, see [DBCExtension.GLOP\\_Add Stored Procedure](#).

For a complete example that creates a GLOP set and uses a UDF to generate the BLOB data for a GLOP mapping, see [C Function Using GLOP Data](#).

## Becoming a Member of a GLOP Set

An external routine can be a member of one GLOP set (although more than one external routine can be a member of the same GLOP set). To make an external routine a member of a GLOP set, use the `USING GLOP SET` clause when you define the external routine.

Here is an example that defines a UDF called `GLOP_Manager` and uses the `USING GLOP SET` clause to make the UDF a member of the `GLOP_Set_1001` GLOP set:

```
CREATE FUNCTION GLOP_Manager(Record_n INTEGER)
RETURNS INTEGER
LANGUAGE C
USING GLOP SET GLOP_Set_1001
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!glop_manager!glop_manager.c';
```

For details on the `USING GLOP SET` clause, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Using GLOP Data in a C/C++ External Routine

An external routine that is a member of a GLOP set must first gain access to the GLOP set before it can use the data in any of the mappings.

To gain access to a GLOP set in a C or C++ external routine, call the `FNC_Get_GLOP_Map` library function. You must call `FNC_Get_GLOP_Map` first before calling any other GLOP functions such as

FNC\_GLOP\_Lock, FNC\_GLOP\_Unlock, FNC\_GLOP\_Map\_Page, or FNC\_GLOP\_Global\_Copy. Also, you should call FNC\_Get\_GLOP\_Map first in each phase of a table function that uses any of the other GLOP routines to initialize the memory pointer for that phase.

Here is a code excerpt that uses the FNC\_Get\_GLOP\_Map function. For a complete example of how to use GLOP data in a C external routine, see [C Function Using GLOP Data](#).

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

struct Company
{
    INTEGER deptid;
    char deptname[30];
};

struct Company_Info
{
    struct Company Dept[20];
}*COMPANY_DETAILS;

void *GLOP_Manager(INTEGER *record_n,
                   INTEGER *result,
                   char sqlstate[6])
{
    int glop_stat;
    GLOP_Map_t *MyGLOP; // Structure GLOP_Map_t is an array of eight
                        // GLOP_ref_t structures, plus some internal
                        // structures. Eight GLOP data references is
                        // the maximum an external routine can map.

    glop_stat = FNC_Get_GLOP_Map(&MyGLOP);

    // Check whether access to the GLOP set is established or not
    if (glop_stat)
    {
        strcpy(sqlstate, "U0001");
        return;
    }

    // Check the first GLOP mapping in the GLOP set
    if (MyGLOP->GLOP[0].GLOP_ptr == NULL)
```

```

{
    strcpy(sqlstate,"U00002");
    return;
}
else
{
    // Use the first GLOP mapping (index 0) here
    COMPANY_DETAILS = MyGLOP->GLOP[0].GLOP_ptr;
    ...
}

...

return;
}

```

For details on the FNC\_Get\_GLOP\_Map C library function, see [FNC\\_Get\\_GLOP\\_Map](#).

## DBCExtension.GLOP\_Add Stored Procedure

Use GLOP\_Add to add GLOP data and GLOP mappings to a GLOP set.

### Required Privileges

An administrator who is adding GLOP data for many users must:

- Have the INSERT privilege on the GLOP\_Set, GLOP\_Map, and GLOP\_Data tables (in the DBCExtension database).
- Include the database the set is being added for in the Set\_Name argument.
- Execute the CREATE GLOP SET statement for each new set being created prior to calling GLOP\_Add.

To add to one of your granted and defined GLOP sets when you do not have the INSERT privilege on the GLOP\_Set, GLOP\_Map, and GLOP\_Data tables (in the DBCExtension database), you must:

- Have the GLOP or CREATE GLOP privilege on the database the GLOP is being added to.
- Be the creator or owner of the GLOP set and have the GLOP MEMBER privilege.

To add an 'RO', 'US' or 'XR' GLOP, the current authorization must own or have one privilege on that role, user or external routine, respectively.

If the Set\_name argument specifies a table name containing the GLOP set information, the caller of the procedure must have the SELECT privilege on the specified table.

For details on GLOP privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

## Syntax

```
CREATE PROCEDURE DBCEExtension.GLOP_Add
  (IN Set_name      VARCHAR(257) CHARACTER SET UNICODE,
   IN Type_g       CHAR(2) CHARACTER SET LATIN,
   IN Association   VARCHAR(257) CHARACTER SET UNICODE,
   IN Data_Name    VARCHAR(128) CHARACTER SET UNICODE,
   IN Shared       CHAR(1) CHARACTERS SET LATIN,
   IN Map_Index    BYTEINT,
   IN Map_Page     VARCHAR(1) CHARACTER SET LATIN,
   IN Mode_g       CHAR(1) CHARACTER SET LATIN,
   IN Disable      CHAR(1) CHARACTER SET LATIN,
   IN Ageing       INTEGER,
   IN Length       INTEGER,
   IN Page         INTEGER,
   IN Data         BLOB(2097088000)
  )
```

## Syntax Elements

### *Set\_name*

Name of a GLOP set or the name of a table, depending on the value of the *Type\_g* argument.

If *Type\_g* is...

- not NULL, then *Set\_name* is the name of the set including the database it is associated with. If the name is not fully qualified, the current default database is assumed. A set name must be a unique TVM name. The system reports an error if the set name is not associated with the GLOP set the user has access to.
- NULL, then *Set\_name* is the name of a table containing the equivalent data in multiple rows to build the GLOP set. If the table is not located in the current default database, then name must be fully qualified.

All other arguments are ignored, so they can be set to NULL.

For details on the definition that the table must have, see [Usage Notes](#).

### *Type\_g*

Type of GLOP, where the valid values are:

- 'SY', meaning System GLOP.
- 'RO', meaning Role GLOP.
- 'US', meaning User GLOP.
- 'SE', meaning Session GLOP.
- 'TR', meaning Transaction GLOP.

- 'RE', meaning Request GLOP.
- 'XR', meaning External Routine GLOP.
- NULL, which means the *Set\_name* argument supplies a table name to contain the data.

For details on the meanings of the GLOP types, see [GLOP Types](#).

### **Association**

Association for particular GLOP types.

If *Type\_g* is...

- 'RO', then the association is the role name.
- 'US', then the association is the user name.
- 'XR', then the association is the external routine name.

If the external routine is not located in the current default database, the name must be fully qualified. For a UDF, the external routine name must be the specific name of the routine.

- any other value, then the association is ignored and should be set to NULL.

If the association name already exists for GLOP data or if the name does not exist in the database, the system reports an error.

### **Data\_Name**

Name assigned to the GLOP data for the given set database. An error is reported if the data name exists for the given database and the Data argument is not set to NULL.

### **Shared**

GLOP mapping attributes, where the valid values are:

- 'Y', meaning that the GLOP mapping is to be shared.

A GLOP mapping can be shared between the GLOP types in the same set or other sets defined in the same database.

- 'N', meaning that the GLOP mapping is not to be shared.

For details on GLOP mapping attributes, see [GLOP Data Attributes](#).

### **Map\_Index**

Position in the map array the GLOP address is to be found in the external routine. Valid values are 0 to 7.

### **Map\_Page**

[for read-only GLOP data] Whether to make this the initial page to map for the index specified by the *Map\_Index* argument. Valid values are:

- 'Y', which specifies to make this the initial page to map for the index specified by the *Map\_Index* argument.
- 'N' or NULL, which specifies to not make this the initial page to map for the specified index.

If there is already a page that is mapped at this index, the system returns an error.

### ***Mode\_g***

GLOP modification attribute. Valid values are:

- 'R', which specifies that it is a read-only GLOP.  
By default, a read-only GLOP is always normal.
- 'W', which specifies that it is a read/write GLOP.
- 'M', which specifies that it is a read/write GLOP that can be globally modified.

For details on GLOP modification attributes, see [GLOP Data Attributes](#).

### ***Disable***

One of the following values:

- 'D', which specifies to disable the data.  
The GLOP page is unavailable until some future date.
- 'E', which specifies to enable the GLOP.

### ***Ageing***

[for read-only GLOP data] Number of minutes to wait before ageing this page out of memory. The valid values are:

- > 0, which specifies to age this page out of memory if this number of minutes elapses and the page is not mapped.
- 0, which specifies not to age this page out of memory.
- -1, which specifies to use the ageing value that was set for the initial page.

If this is the initial page, the ageing value is set to 0.

### ***Length***

Mapped length of the GLOP being created here. This argument is not relevant if the GLOP named by *Data\_Name* already exists. The rest of the description assumes the GLOP data is being created for the first time. The length field can be independent of the actual length of the supplied data. If the data is longer, it will be truncated when mapped. If the data is shorter, the remaining length will be zeroed when mapped. A NULL will cause an error return for a new GLOP data. For read-only GLOP data, all pages have the same length.

A value of -1 for read-only GLOP data sets the length to the length of the biggest GLOP page. A value of -1 will for read/write or globally modifiable GLOP data sets the length to the length of the data. Data must be specified in that case.

### **Page**

[for read-only GLOP data] Number of the GLOP page to be created. GLOP page numbers must be sequential, where the value of the initial page is 1.

If the GLOP mode is not read-only, this parameter is ignored. The system reports an error if a gap exists in the page number sequence or if the page already exists.

### **Data**

Actual data to be loaded into the system table. The data is only supplied when the GLOP Data\_Name is being added the very first time to a GLOP set. For all subsequent references via the Data\_Name field, that use this data in another set or the same set, the field is set to NULL. The data can be specified either directly or it can be generated by specifying a UDF that returns a BLOB to this argument. The data can be NULL. If that is the case a GLOP specified by the Length parameter will be zeroed and mapped. Passing NULL for read-only GLOP data results in an error.

## **Usage Notes**

Use this procedure to define new GLOP data and its attributes and mapping for a GLOP set. It cannot be used to drop or replace existing data in the set. Call GLOP\_Remove to drop GLOP data and GLOP\_Change if data is to be changed. The procedure will expect the GLOP set to exist already in the database it is being created for, so issue a CREATE GLOP SET statement using the name of the new GLOP set to create, prior to calling this procedure the first time for the given GLOP set name.

All references to the same GLOP data must be associated with the same database. Referencing the same GLOP data in multiple sets is done by specifying the Data\_Name and the Data the first time the GLOP is being added. For all subsequent referenced additions to the same data, the Data\_Name must be supplied and the Data argument must be NULL. The first time the GLOP data is added determines what its contents are to be. If the GLOP has no data, setting the Data argument to NULL for the first GLOP add, will accomplish that feat.

All parameters are checked for valid combinations of data and consistency. In some cases, the supplied information is ignored. For example, an ageing value is ignored for read/write GLOP data since it has no meaning (but it causes no harm to supply a value). If there is an inconsistency in any of the parameter settings that are not called out specifically in the parameter section, the system reports an error.

Do not call this procedure from within a transaction because it creates its own transaction, possibly more than one and might execute DDL statements. The procedure is designed to run from a Teradata mode session.

If the Type\_g field is NULL it expects the Set\_name argument to contain a table name that will contain the data to be added to the GLOP set. The table must have the following definition. It can be a temporary table.



```

CREATE TABLE GLOP_Add_Table      /* the name can be anything */
  (Sequence      INTEGER NOT NULL,
   Set_name      VARCHAR(257) CHARACTER SET UNICODE,
   Type_g        CHAR(2) CHARACTER SET LATIN,
   Association    VARCHAR(257) CHARACTER SET UNICODE,
   Data_Name     VARCHAR(128) CHARACTER SET UNICODE,
   Shared        CHAR(1) CHARACTERS SET LATIN,
   Map_Index     BYTEINT,
   Map_Page      VARCHAR(1) CHARACTER SET LATIN,
   Mode_g        CHAR(1) CHARACTER SET LATIN,
   Disable       CHAR(1) CHARACTER SET LATIN,
   Ageing        INTEGER,
   Length        INTEGER,
   Page          INTEGER NOT NULL,
   Data          BLOB(2097088000))
PRIMARY INDEX (Sequence);

```

The descriptions of the defined columns are identical to the stored procedure version and have the same name. The only additional column is the Sequence column which must contain a number increasing in the order the data is to be applied.. The data will be sorted in ascending order on Sequence. An error will be reported if the sequence number does not increase or there is a gap in the Page numbers, which must be in ascending order for multipage read-only GLOP data. More than one set can be added using this technique.

### Example: DBCExtension.GLOP\_Add Stored Procedure

Create an XML document containing parsing rules for PDF documents. This is the first time.

```

CREATE GLOP SET Web_XML_Document_Markup;

CALL DBCExtension.GLOP_Add(
  'Web_XML_Document_Markup', -- the set name
  'SE',                       -- this is session GLOP data
  NULL,                      -- Not relevant for session GLOP data
  'PDF_Mode',                -- named the GLOP data
  NULL,                      -- shared flag not relevant
  0,                         -- Put in map index 0
  'Y',                       -- Make this the initial map page
  'R',                       -- Read-only
  'E',                       -- Not disabled
  10,                        -- Ageing is 10
  -1,                        -- Set GLOP length to data length
  1,                         -- Creating page one
  Gen_XML_PDF_Markup(...)); -- a UDF creates the GLOP data

```

This is read-only GLOP data so it is not necessary to make it a session type of GLOP, but it might be useful for dependency changes in the future, making it easy to replace it on session boundaries.

## DBCExtension.GLOP\_Remove Stored Procedure

Use GLOP\_Remove to delete an entire GLOP set or remove individual components from a GLOP set.

Before you can use the DROP GLOP SET statement to drop a GLOP set definition, you must use GLOP\_Remove to delete the GLOP set and its data.

### Required Privileges

An administrator who is removing GLOP data for many users must have the DELETE privilege on the DBCExtension.GLOP\_Set, DBCExtension.GLOP\_Map, and DBCExtension.GLOP\_Data tables. The Set\_Name argument must include the database the set is being removed for.

To remove your own set when you do not have the DELETE privilege on the DBCExtension.GLOP\_Set, DBCExtension.GLOP\_Map, and DBCExtension.GLOP\_Data tables, you must:

- Have the GLOP or DROP GLOP privilege on the database from which the GLOP is being removed.
- Be the creator or owner of the GLOP set and have the GLOP MEMBER privilege.

To remove an 'RO', 'US' or 'XR' GLOP, the current authorization must at least own or have one privilege on that role, user, or external routine.

If the Set\_name argument passes in a table name containing the GLOP set information, the user running the procedure must have the SELECT privilege on the specified table.

For details on GLOP privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

### Syntax

```
REPLACE PROCEDURE DBCExtension.GLOP_Remove
  (IN Set_name      VARCHAR(257) CHARACTER SET UNICODE,
   IN Type_g        CHAR(2) CHARACTER SET LATIN,
   IN Association    VARCHAR(257) CHARACTER SET UNICODE,
   IN Map_Index      BYTEINT,
   IN Options        CHAR(1) CHARACTER SET LATIN
  )
```

### Syntax Elements

#### Set\_name

Name of a GLOP set or the name of a table, depending on the value of the *Type\_g* argument.

If *Type\_g* is...

- not NULL, then *Set\_name* is the name of the set from which components are to be removed. If the name is not fully qualified, the current default database is assumed. If the name does not exist, the system reports an error.

- NULL, then *Set\_name* is the name of a table containing the equivalent data in multiple rows of the GLOP set or sets to remove. If the table is not located in the current default database, then the name must be fully qualified.

All other arguments are ignored, so they should be set to NULL.

For details on the definition that the table must have, see [Usage Notes](#).

### ***Type\_g***

Type of GLOP, where the valid values are:

- 'SY', meaning System GLOP.
- 'RO', meaning Role GLOP.
- 'US', meaning User GLOP.
- 'SE', meaning Session GLOP.
- 'TR', meaning Transaction GLOP.
- 'RE', meaning Request GLOP.
- 'XR', meaning External Routine GLOP.
- NULL, which means the *Set\_name* argument supplies a table name to contain the data.

For details on the meanings of the GLOP types, see [GLOP Types](#).

### ***Association***

Association for particular GLOP types. NULL is a valid value for these GLOP types if the Options argument has a value of 'T'.

If *Type\_g* is...

- 'RO', then the association is the role name.
- 'US', then the association is the user name.
- 'XR', then the association is the external routine name.

If the external routine is not located in the current default database, the name must be fully qualified. For a UDF, the external routine name must be the specific name of the routine.

- any other value, then the association is ignored and should be set to NULL.

If the association name does not exist in the database, the system reports an error.

### ***Map\_Index***

Mapped index position of the GLOP to be removed.

If *Type\_g* is...

- 'RO', 'US', or 'XR', then the map index entry is set to NULL when the last one of this type is removed.
- any other value, then the GLOP index entry is removed.

## Options

How much to remove. The valid values are:

- 'A', which specifies to remove all GLOP set entries for the GLOP set.

The value of the *Type\_g* argument is ignored (because the entire set will be removed) unless it is NULL, which means that the *Set\_name* argument supplies a table name to contain the data.

- 'T', which specifies to remove a GLOP set entry of the designated type.

If the value of the *Type\_g* argument is 'RO', 'US', or 'XR', and the value of the *Association* argument is NULL, then all GLOP data of that type is removed for the given set. If the *Association* argument is not NULL, then that specific GLOP data is removed for the given set.

Individual pages of a read-only GLOP cannot be removed.

## Usage Notes

Use the GLOP\_Remove procedure to remove an existing GLOP set either in whole or in parts. To permanently delete all traces of the entire GLOP set, use the DROP GLOP SET statement after calling the GLOP\_Remove procedure.

All parameters are checked for valid combinations of data and consistency. In some cases, the supplied information will be ignored. For example when removing a 'SE' GLOP set the Association argument has no meaning. If the parameters are incorrect or are used out of context, the system reports an error.

The procedure should not be called from within a transaction, because it creates its own transaction, possibly more than one and might execute DDL statements. The procedure is designed to run from a Teradata mode session.

When all references to a specific GLOP data type are removed the data is deleted also. For a non shared GLOP that occurs when the entry of the specific type is deleted from the GLOP set. For a shared GLOP this occurs when the last GLOP set reference to it is deleted.

If the *Type\_g* field is NULL it expects the *Set\_name* parameter to contain a table name that will contain the data to be removed from the GLOP set. The table must have the following definition. It can be a temporary table.

```
CREATE TABLE GLOP_Remove_Table    /* the name can be anything */
  (Sequence      INTEGER NOT NULL,
   Set_name      VARCHAR(257) CHARACTER SET UNICODE,
   Type_g        CHAR(2) CHARACTER SET LATIN,
   Association    VARCHAR(257) CHARACTER SET UNICODE,
   Map_Index     BYTEINT,
   Options       CHAR(1) CHARACTER SET LATIN)
PRIMARY INDEX (Sequence);
```

The descriptions of the defined columns are identical to the stored procedure version and have the same name. The only additional column is the Sequence column that must contain a number increasing in the order the data is to be applied. If the sequence number does not increase, the system reports an error.

### Example: DBCExtension.GLOP\_Remove Stored Procedure

Remove the entries for the 'Auditor' role, which is no longer being used.

```
CALL DBCExtension.GLOP_Remove(
  'Accounting',  -- the accounting GLOP set
  'RO',          -- removing a role
  'Auditor',     -- role being removed from the set
  2,            -- mapped to index 2
  'T');         -- remove only specified type
```

If this is the last one, then execute the DROP GLOP SET statement:

```
DROP GLOP SET Accounting;
```

## DBCExtension.GLOP\_Change Stored Procedure

Use GLOP\_Change to change the GLOP data and any of the following settings:

- GLOP\_Data for existing pages
- Ageing time for read-only GLOP data
- Disable field of GLOP data
- Length of the GLOP data to map

### Required Privileges

An administrator who is changing GLOP data for many users must have the UPDATE privilege on the GLOP\_Set, GLOP\_Map and GLOP\_Data tables (in the DBCExtension database). The Set\_Name argument must be fully qualified to include the database the GLOP set is being changed for.

To change your created GLOP sets when you do not have the UPDATE privilege on the GLOP\_Set, GLOP\_Map and GLOP\_Data tables (in the DBCExtension database), you must:

- Have the GLOP or DROP GLOP privilege on the database for which the GLOP is being changed.
- Be the creator or owner of the GLOP set and have the GLOP MEMBER privilege.

To change an 'RO', 'US' or 'XR' GLOP the current authorization must at least own or have one privilege on the respective role, user, or external routine.

If the table name containing the GLOP set information is being passed in the Set\_name argument the user running the procedure must have SELECT privileges to the specified table.

You cannot use GLOP\_Change to remove or add GLOP data to an existing set.

## Syntax

```

REPLACE PROCEDURE DBCEExtension.GLOP_Change
  (IN Set_name      VARCHAR(257) CHARACTER SET UNICODE,
   IN Type_g        CHAR(2) CHARACTER SET LATIN,
   IN Dependency     CHAR(4) CHARACTER SET LATIN,
   IN Association    VARCHAR(257) CHARACTER SET UNICODE,
   IN Data Name     VARCHAR(128) CHARACTER SET UNICODE,
   IN Map_Index      BYTEINT,
   IN Disable        CHAR(1) CHARACTER SET LATIN,
   IN Ageing         INTEGER,
   IN Length         INTEGER,
   IN Page           INTEGER,
   IN Data           BLOB(2097088000)
  )

```

## Syntax Elements

### *Set\_name*

Name of a GLOP set or the name of a table, depending on the value of the *Type\_g* argument.

If *Type\_g* is...

- not NULL, then *Set\_name* is the name of the GLOP set to change. If the name is not fully qualified, the current default database is assumed. If the name does not exist, the system reports an error.
- NULL, then *Set\_name* is the name of a table containing the equivalent data in multiple rows of the GLOP set or sets to change. If the table is not located in the current default database, then the name must be fully qualified.

All other arguments are ignored, so they should be set to NULL.

For details on the definition that the table must have, see [Usage Notes](#).

### *Type\_g*

Type of GLOP to change. The valid values are:

- 'SY', meaning System GLOP.
- 'RO', meaning Role GLOP.
- 'US', meaning User GLOP.
- 'SE', meaning Session GLOP.
- 'TR', meaning Transaction GLOP.
- 'RE', meaning Request GLOP.
- 'XR', meaning External Routine GLOP.
- NULL, which means the *Set\_name* argument supplies a table name to contain the data.

For details on the meanings of the GLOP types, see [GLOP Types](#).

### **Dependency**

When the change to a particular GLOP becomes active in memory. It contains GLOP replace dependency information.

You must pass a value of NULL, which means no dependency, change immediately. Other values are reserved for future releases.

### **Association**

Association for particular GLOP types.

If *Type\_g* is...

- 'RO', then the association is the role name.
- 'US', then the association is the user name.
- 'XR', then the association is the external routine name.

If the external routine is not located in the current default database, the name must be fully qualified. For a UDF, the external routine name must be the specific name of the routine.

- any other value, then the association is ignored and should be set to NULL.

If the association name does not exist in the database, the system reports an error.

### **Data\_Name**

Name of the GLOP data to change. If the name does not exist, the system reports an error.

### **Map\_Index**

Position in the map array the GLOP address is to be set in the external routine. Valid values are 0 to 7.

### **Disable**

Whether to enable or disable the data. The valid values are:

- 'D', which specifies to disable the data.
- 'E', which specifies to enable the GLOP.
- NULL, which specifies to not change the value.

### **Ageing**

Whether to age a read-only GLOP out of memory when not mapped for the given number of minutes. The valid values are:

- > 0, which specifies to age the GLOP out of memory when not mapped for the given number of minutes.

- 0, which specifies to not age the GLOP out of memory.
- NULL, which specifies to not change the value.

### **Length**

Mapped length of the GLOP data, independent of the actual length of the data. If the data is longer, it will be truncated when mapped. If the data is shorter, the remaining length will be zeroed when mapped. The valid values are:

- > -1, which specifies to set the length to the specified value.
- NULL, which specifies to retain the current length.
- -1, which specifies to set the length to the size of the largest GLOP page for the set ID, set type, and map index.

### **Page**

Number of the GLOP page to be changed. If the value is NULL, no data will be changed. If the GLOP page does not exist, the system reports an error. This value is only needed for read-only GLOP data; however, a value of 1 can be specified for read/write or globally modifiable GLOP data.

### **Data**

Actual data to be changed. If this value is NULL, the data is not changed. If a zero length byte string is passed, the data is deleted but the GLOP page will be retained. The data can be specified either directly or it can be generated by specifying a UDF that returns a BLOB to this argument.

## **Usage Notes**

Use this procedure to change the attributes or data of a GLOP set. Do not use it to remove or add additional GLOP data to the set. The changes cause all changed GLOP data to be replaced in memory.

If the Type\_g argument is NULL, the Set\_name argument must specify a table name that contains the data to be changed in the GLOP set. The table must have the following definition. It can be a temporary table.

```
CREATE TABLE GLOP_Change_Table
(Sequence      INTEGER NOT NULL,
 Set_name      VARCHAR(257) CHARACTER SET UNICODE,
 Type_g       CHAR(2) CHARACTER SET LATIN,
 Dependency    CHAR(4) CHARACTER SET LATIN,
 Association   VARCHAR(128) CHARACTER SET UNICODE,
 Data_Name     VARCHAR(128) CHARACTER SET UNICODE,
 Map_Index     BYTEINT,
 Disable      CHAR(1) CHARACTER SET LATIN,
 Ageing        INTEGER,
```



```

    Length      INTEGER,
    Page        INTEGER,
    Data        BLOB(2097088000))
PRIMARY INDEX (Sequence);

```

The descriptions of the defined columns are identical to the stored procedure arguments and have the same name. The only additional column is the Sequence column that must contain a number increasing in the order the data is to be applied. The system reports an error if the sequence does not increase.

Changes do not become permanent until there is a change in the Set\_name or the last row has been processed. It works this way so that dependency information or index mappings can be applied at once to prevent a partial change from causing GLOP mapping to take place at the wrong time. However, all changes are rolled back when there is an error.

Do not call GLOP\_Change from within a transaction because it creates its own transaction, possibly more than one, and might execute DDL statements.

The procedure is designed to run from a Teradata mode session.

### Example: DBCExtension.GLOP\_Change Stored Procedure

This simple example updates an 'Engineering' set GLOP with new update aluminum stress/strength data, for various alloys of aluminum. The data already exists; it is simply being replaced with newer data. The data is needed by the 'Stress01' UDF, which uses the data to do structural analysis on an engineering project. This is read-only GLOP data and it can simply be replaced for the next transaction when 'Stress01' is called.

```

CALL DBCExtension.GLOP_Change(
    'Engineering',      -- Set name
    'XR',               -- Type: UDF
    NULL,               -- No dependency
    'Stress01',         -- UDF that uses data
    'Aluminum_Stress_Tables', -- the name of GLOP
    NULL,               -- No change to map index
    NULL,               -- No change to disable setting
    NULL,               -- Do not change ageing value
    -1,                 -- Set GLOP length to data size
    1,                  -- Change page 1
    Engin_Strs_Gen('Al') -- UDF to create the stress data
);

```

### DBCExtension.GLOP\_Report Stored Procedure

Use the GLOP\_Report procedure to output specific information about a particular GLOP set or all GLOP sets. It does not display the GLOP data. It provides a complete report consisting of as many result sets as needed depending on what is requested.

## Required Privileges

An administrator who is reporting on any desired GLOP must have the SELECT privilege on the DBCExtension.GLOP\_Set, DBCExtension.GLOP\_Map, and DBCExtension.GLOP\_Data tables.

To report on your own set when you do not have the SELECT privilege on the DBCExtension.GLOP\_Set, DBCExtension.GLOP\_Map, and DBCExtension.GLOP\_Data tables, you must be the owner or creator of the GLOP or have the GLOP MEMBER privilege on the GLOP.

## Syntax

```
REPLACE PROCEDURE DBCExtension.GLOP_Report
  (IN Options CHAR(1),
   IN FindName VARCHAR(257))
```

## Syntax Elements

### Options

What to report. Valid values are as follows. Any other value generates an error.

- 'A', which specifies to report on all sets.

The user must have sufficient privileges to report on all GLOP sets; if not, it reports on the sets that belong to the user running the report.

- 'D', which specifies to report on a specific named data GLOP.
- 'S', which specifies to report on a specific set.

### FindName

the name of the object to report information on for the given options.

For option 'A' this argument is a database name or a database name followed by the set name using dot notation (*dbname.setname*). A NULL in this field will report on all sets provided user has sufficient privileges.

For option 'S' this argument is the set name when the database is not specified or database name and set name using dot notation.

For option 'D' this argument is the data name when the database is not specified or the database name and data name using dot notation (*dbname.dataname*).

## Usage Notes

Use this procedure to provide detailed information of specific GLOP sets or all of them.

The following information is reported:

- A result set listing all the set numbers with their associated name in set name ascending order
  - GLOP\_Set\_ID

- GLOP\_SetName (ascending order)
- A result set showing the details for each set
  - GLOP\_Set\_ID ( first sort field in ascending order)
  - GLOP\_Type (second sort field in ascending order )
  - GLOP\_Dependency
  - GLOP\_Association (third sort field in ascending order)
  - Role, User, or External Routine ID
  - GLOP\_Mode
  - GLOP\_Disable
  - GLOP\_Ageing
  - GLOP\_Length
  - GLOP\_Pages
  - GLOP\_Data\_Ref (fourth sort field in ascending order)
  - GLOP\_Page number (fifth sort field in ascending order)
- A result set showing the data reference ID and the corresponding name, sorted in data reference id order
  - GLOP\_Data\_Ref (ascending order)
  - GLOP\_Data\_Name
- A result set showing the mapping information for the Set
  - GLOP\_Set\_ID (first sort field in ascending order)
  - Index of the entry (second sort field in ascending order)
  - The GLOP\_Type
  - GLOP\_Dependency
  - The GLOP\_MapX shared or not shared flag
  - The GLOP\_ModeX mode of the GLOP
  - Initial page mapped: GLOP\_PageX
- A result set showing the user association ID and the corresponding name for the user in association id ascending order
  - GLOP\_Association
  - GLOP\_AssociationName
- A result set showing the role association ID and the corresponding name for the role in association id ascending order
  - GLOP\_Association
  - GLOP\_AssociationName
- A result set showing the external routine association ID and the corresponding external routine name and database in association ID ascending order

- GLOP\_Association
- GLOP\_AssociationName

### Example: DBCExtension.GLOP\_Report Stored Procedure

Output a report for the 'sales\_division' GLOP set by a specific user.

```
CALL DBCExtension.GLOP_Report('S', 'sales_division');
```

## System Disk Space

The system disk space must have enough free space to contain the mapped GLOP data.

## Global Configuration Settings

Teradata provides a utility called *cufconfig* that you can use to view or modify the following global configuration settings for GLOP data.

Global Configuration Setting	cufconfig Field
Timeout to wait for a GLOP lock to free up	GLOPLockTimeout
Timeout to hold a GLOP lock	GLOPLockWait
Maximum size of GLOP data	MaximumGLOPSize
Maximum amount of memory one vproc can use for GLOP mappings	MaximumGLOPMem
Directory path for GLOP mappings	GLOPMapMemPath
Maximum number of GLOP pages that can be allocated for a read-only GLOP	MaximumGLOPPages

For information on using the *cufconfig* utility to change global configuration settings for GLOP data, see *Teradata Vantage™ - Database Utilities*, B035-1102.

# Administration

The following sections provide information specific to the administration of UDFs, UDMs, and external stored procedures (collectively called *external routines*).

Administration includes tasks that you perform in support of external routine development and maintenance.

## System Requirements

Before you can install or invoke external routines, your system must meet certain requirements.

IF you are developing ...	THEN your system must have ...
UDFs, UDMs, or external stored procedures written in C or C++	a C or C++ compiler on the server that Teradata uses to compile the external routine.
a C or C++ external stored procedure that uses CLIV2 to directly execute SQL	a current version of CLIV2 on all PE nodes that will execute the external stored procedure.
Java UDFs or external stored procedures	<p>a C compiler on the server that you use to execute the DIPSQJ script from the DIP utility.</p> <p>the Teradata JRE package on all nodes that will execute the Java UDF or external stored procedure.</p> <p>The JRE installation path is a global configuration setting that you can view or modify using the <i>cufconfig</i> utility. For information, see <i>Teradata Vantage™ - Database Utilities</i>, B035-1102.</p>
a Java external stored procedure that executes SQL	<p>a Teradata Driver for the JDBC Interface on the nodes that will execute the external stored procedure.</p> <p>For more information on the Teradata Driver for the JDBC Interface, see <i>Teradata JDBC Driver Reference</i>, available at <a href="https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html">https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html</a>.</p>

## Global Configuration Settings

Teradata provides a utility called *cufconfig* that you can use to view or modify global configuration settings for UDFs, UDMs, and external stored procedures.

For example, you can use *cufconfig* to view the path to libraries required by C or C++ external stored procedures that use CLIV2 to execute SQL or to view the installation path to the Java Development Kit (JDK) and Java Runtime Environment (JRE) for Java external routines.

For information on using the *cufconfig* utility to change global configuration settings for UDFs, UDMs, and external stored procedures, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Protected Mode Process and Server Administration for C/C++ External Routines

Executing C or C++ external routines in protected mode requires additional administration.

The database installation process creates a local, operating system-native user with the name 'tdatuser' on each node the database runs on. The 'tdatuser' ID is associated with two default processes per AMP vproc and PE vproc for running external routines in protected mode.

Follow these rules to guarantee that your external routines and database function properly:

- Do not delete 'tdatuser'.
- Do not add a password for 'tdatuser'.

This prevents logon to the user on the system.

- If you think you need to make changes, please contact the Teradata Support Center first.

In addition to creating 'tdatuser', the installation process also creates a new group on each node called 'tdatudf'. The preceding admonishments apply to the 'tdatudf' group: do not change it.

The 'tdatuser' user has no special privileges on the system. No one should be able to log on as 'tdatuser'.

If an external routine that runs in protected mode needs to access system resources, to open a file for example, you must set the appropriate access privileges to include 'tdatuser'.

By default, each vproc has only two protected mode servers that are shared among all transactions executing protected mode external routines. To increase the number of protected mode servers per vproc, use the *cufconfig* utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

A table function that executes in protected mode reserves a server process for the duration of the step that executes the table function. It is important that the number of protected mode servers that you allocate is greater than the number of simultaneous queries from different sessions that you plan to run. If you allocate too few, then some will have to wait. If all the servers are used up, an undetected deadlock could also occur.

## Server Administration for Java External Routines

### Hybrid Server Administration for Java UDFs

Executing Java UDFs in protected mode (where the CREATE FUNCTION or REPLACE FUNCTION statement does not include the EXTERNAL SECURITY clause) requires additional administration.

The database installation process creates a local, operating system-native user with the name 'tdatuser' on each node the database runs on.

Each node can have one hybrid server that runs under the authorization of the 'tdatuser' operating system user. The server provides multiple threaded execution of protected mode Java UDFs to all AMPs and PEs on the node.

Follow these rules to guarantee that your external routines and database function properly:

- Do not delete 'tdatuser'.

- Do not add a password for 'tdatuser'.

This prevents logon to the user on the system.

- If you think you need to make changes, please contact the Teradata Support Center first.

In addition to creating 'tdatuser', the installation process also creates a new group on each node called 'tdatudf'. The preceding admonishments apply to the 'tdatudf' group: do not change it.

The 'tdatuser' user has no special privileges on the system. No one should be able to log on as 'tdatuser'.

If an external routine that runs in protected mode needs to access system resources, to open a file for example, you must set the appropriate access privileges to include 'tdatuser'.

The hybrid server uses a global configuration setting to determine the maximum number of threads for running Java UDFs. To view or change global configuration settings for external routines, use the *cufconfig* utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Java Secure Server Administration

The database uses a secure server mechanism to execute the following:

- Java external stored procedures
- Java UDFs where the CREATE FUNCTION or REPLACE FUNCTION statement includes the EXTERNAL SECURITY clause

Executing Java external routines in secure mode requires additional administration.

IF the external routine is ...	THEN ...
an external stored procedure where the CREATE PROCEDURE or REPLACE PROCEDURE statement does not include the EXTERNAL SECURITY clause	the external stored procedure runs in protected execution mode as a separate process under 'tdatuser', a local operating system user that the database installation process creates. Follow the administration guidelines for the 'tdatuser' user from the preceding section, "Hybrid Server Administration for Java UDFs".
a UDF or external stored procedure where the CREATE/REPLACE FUNCTION or CREATE/REPLACE PROCEDURE statement includes the EXTERNAL SECURITY clause	Vantage uses a separate secure server to execute the external routine under the authorization of a specific native operating system user established by a CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement. If an external routine that runs in secure mode needs to access system resources, to open a file for example, you must set the appropriate access privileges to include the user established by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement.

The database uses a global configuration setting to determine the maximum number of secure servers that can execute external routines. To view or change global configuration settings for external routines, use the *cufconfig* utility. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Finding Unprotected Mode UDFs and Changing the Execution Mode

Teradata provides BTEQ scripts that you can use to generate a list of UDFs that currently execute in unprotected mode and change the execution mode.

The scripts are in the `etc` directory of the Teradata software distribution. To identify this directory, enter the following on the command line:

```
pdepath -e
```

To run the scripts, you must logon to BTEQ as user DBC. For example:

```
bteq .logon dbc,dbc_password
```

where `dbc_password` is the password for user DBC.

IF you want to ...	THEN use this script ...
get a list of unprotected mode UDFs on the system	<code>udflist.bteq</code> . The script creates a file called <code>udflist.sql</code> in the current working directory. You must have write permission on the current working directory.
read a list of UDFs and change the execution mode to protected	<code>udf_u2p.bteq</code> . The script reads a list of UDFs from a file called <code>udf_u2p.sql</code> and changes the execution mode to protected. To create the contents of the <code>udf_u2p.sql</code> file, use the list of UDFs that the <code>udflist.bteq</code> script generates in the <code>udflist.sql</code> file.
read a list of UDFs and change the execution mode to unprotected	<code>udf_p2u.bteq</code> . The script reads a list of UDFs from a file called <code>udf_p2u.sql</code> and changes the execution mode to unprotected. To create the contents of the <code>udf_p2u.sql</code> file, use the list of UDFs that the <code>udflist.bteq</code> script generates in the <code>udflist.sql</code> file.

For example, suppose you installed two UDFs on the `cs` database that execute in unprotected mode: `imult` and `isqr`. If you run `udflist.bteq`, it produces a file called `udflist.sql` in the current working directory with the following contents:

```
/* ----- */
/* List Of Unprotected Mode UDFs */
/* ----- */
'6373'xn.'696D756C74'xn /* cs.imult */
'6373'xn.'69737172'xn /* cs.isqr */
```



To change the execution mode of the UDFs to protected, copy the UDF names from `udflist.bteq` into a file called `udf_u2p.sql` in the current working directory:

```
'6373'xn.'696D756C74'xn /* cs.imult */
'6373'xn.'69737172'xn /* cs.isqr */
```

When you run `udf_u2p.bteq`, it reads the contents of `udf_u2p.sql` from the current working directory.

## File System Cleanup

Each time you create, replace, or drop a C or C++ external routine, Vantage creates a new dynamic linked library (DLL or SO) associated with the database in which the external routine resides and distributes it to all nodes.

Under normal circumstances, the database deletes the older version of the library, if one exists, provided that the older version of the library is not being used at the time (for example, no running transactions are executing any of the external routines in the older version of the library).

If your environment is one where you frequently create or replace external routines, and seldom execute any of the external routines, you may inadvertently accumulate libraries that the database cannot delete.

If you accumulate enough libraries to fill up the file system, an attempt to create or replace an external routine will likely result in an error message that the system cannot create a new DLL for the UDF/external stored procedure/UDM.

If you get this message, check the subdirectories where the database saves the libraries:

```
<tdconfig>/udflib
```

where `<tdconfig>` is the configuration directory of the Teradata software distribution. To identify this directory, enter the following on the command line:

```
pdepath -c
```

Typically, each subdirectory, with a name such as `tdbs_1053`, contains one dynamic linked library.

The best practice to clean up old libraries is to restart the database. Deleting libraries manually is not recommended and may cause problems.

## Registering and Distributing JAR and ZIP Files for Java External Routines

After you write, test, and debug the source code that implements a Java UDF or external stored procedure (or both), you place the resulting class or classes in a JAR or ZIP file (collectively called *archive* files). Next, you must call the `SQLJ.INSTALL_JAR` external stored procedure to:

- Register the archive file and its classes with the database
- Distribute the archive file to all nodes of a system

- Create an SQL identifier for the archive file that you use in the CREATE/REPLACE PROCEDURE or CREATE/REPLACE FUNCTION statement to define the Java external routine

When you call SQLJ.INSTALL\_JAR, you register the archive file with the default database. Later, when you use the CREATE/REPLACE PROCEDURE or CREATE/REPLACE FUNCTION statement to define an external routine that is implemented by a class in the archive file, you must define the external routine in the same database in which the file was registered.

If you have other JAR or ZIP files that contain application classes required by an external routine, you can use SQLJ.INSTALL\_JAR to register those files with the same database. Then, you can use SQLJ.ALTER\_JAVA\_PATH to specify that the search path of classes include classes from other JAR or ZIP files registered with the same database.

If you call SQLJ.INSTALL\_JAR, you must have sufficient privileges on the default database to use the CREATE/REPLACE PROCEDURE or CREATE/REPLACE FUNCTION statement (or both). If other users need to define an external routine that is implemented by a class in the archive file, they must have the CREATE EXTERNAL PROCEDURE or CREATE FUNCTION privilege on the database in which the archive file was registered.

A recommended practice is to register all archive files and define all Java external routines in one database and then grant access to the external routines to users who need to execute them.

## Before You Begin

Before you can register a JAR or ZIP file for an external routine, verify the SQLJ system database contains the *INSTALL\_JAR* external stored procedure and that you have sufficient privileges on the procedure and on the default database.

1. Use the following SQL statement to view the objects in the SQLJ system database:

```
HELP DATABASE SQLJ;
```

2. If the information that HELP DATABASE returns does not include the *INSTALL\_JAR* external stored procedure, execute the DIPSQJ script using the Database Initialization Program (DIP) utility.

For more information on DIP, see *Teradata Vantage™ - Database Utilities*, B035-1102.

3. Verify you have the EXECUTE privilege on the *INSTALL\_JAR* external stored procedure.
4. If the default database is not the database with which you want to register the JAR or ZIP file, use the DATABASE statement to change the default database.
5. Verify you have the CREATE EXTERNAL PROCEDURE or CREATE FUNCTION privilege on the default database.

## SQLJ.INSTALL\_JAR External Stored Procedure

Use the SQLJ.INSTALL\_JAR external stored procedure to register an archive file (a file with a .jar or .zip file extension) and distribute the archive file to all nodes of a database system.

## Syntax

```
REPLACE PROCEDURE SQLJ.INSTALL_JAR
  (IN locspec VARCHAR(1000),
   IN jarname VARCHAR(128) CHARACTER SET UNICODE,
   IN deploy INTEGER)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SL!xsplib';
```

## Syntax Elements

### *locspec*

Location of the archive file to register.

The first two characters of the *locspec* string specify whether the archive file is on the client or on the database server. If the first two characters are...

- CJ, then the .jar or .zip file is on the client.
- SJ, then the .jar or .zip file is on the database server.

The third character of the *locspec* string is a delimiter that you choose to separate the first two characters from the remaining characters in the *locspec* string.

The remaining characters in the *locspec* string specify the archive file path.

If the archive file is on the...

- client, then the file path is a client-interpreted path that specifies the location and name of the file, including the .jar or .zip file extension.
- database server, then the file path can be a full or relative path that specifies the location and name of the file, including the .jar or .zip file extension.

If the file path is relative, the full path to the archive file is formed by appending the relative path to the default path for source code on the server.

To determine the default path for source code, use the -o option of the *cufconfig* utility and find the setting for the SourceDirectoryPath field:

```
cufconfig -o
```

For more information on *cufconfig*, see *Teradata Vantage™ - Database Utilities*, B035-1102.

### *jarname*

an SQL identifier for the archive file, enclosed in apostrophes ( ' ).

The identifier that you use here to register the archive file is the same name that you specify later in the CREATE/REPLACE PROCEDURE or CREATE/REPLACE FUNCTION statement that you use to define the Java external routine.

The identifier must follow the rules for naming database objects without enclosing the names in double quotation marks:

- They may only include the following characters:
  - Uppercase or lowercase letters (A to Z and a to z)
  - Digits (0 through 9)
  - The special characters HYPHEN-MINUS (-), FULL STOP (.), DOLLAR SIGN (\$), NUMBER SIGN (#), and LOW LINE (\_)
  - (Using the - and . special characters is an extension to the rules for naming database objects without enclosing the names in double quotation marks, and is only allowed for this type of identifier.)
- They must not begin with a digit.
- They must not be a Teradata-reserved word.

For details on Teradata-reserved words, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

- The maximum length for *jarname* is 128 characters.

Additionally, the identifier cannot be a name that is reserved for JAR files that are part of the Teradata installation, where the restriction applies to any combination of uppercase and lowercase letters:

- javFnc
- terajdbc4
- tdgssjava
- tdgssconfig

For details on database object names, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

### ***deploy***

An integer value that is reserved for future use.

For now, the value of *deploy* must be zero.

### **Example: SQLJ.INSTALL\_JAR External Stored Procedure**

Suppose the default path for source code on a Linux server is set to /etc/opt/teradata/tdconfig /Teradata/tdbs\_udf/usr/ and you have a file called accounts.jar in /etc/opt/teradata/tdconfig /Teradata/tdbs\_udf/usr/java\_xsp.

The following statements register the accounts.jar file with the JXSP database, create an SQL identifier called Accounts\_JAR for the file, and distribute the file to all nodes:

```
DATABASE JXSP;
CALL SQLJ.INSTALL_JAR('SJ!java_xsp/accounts.jar', 'Accounts_JAR', 0);
```

Consider a ZIP file called reports.zip on a Linux client in the directory /tmp/java\_xsp.

The following statements register the file with the JXSP database, create an SQL identifier called Reports-ZIP for the file, and distribute the file to all nodes:

```
DATABASE JXSP;
CALL SQLJ.INSTALL_JAR('CJ?/tmp/java_xsp/reports.zip', 'Reports-ZIP', 0);
```

## Related Information

For details on how you use the *jarname* SQL identifier when you define a Java external routine, see CREATE PROCEDURE, REPLACE PROCEDURE, CREATE FUNCTION, and REPLACE FUNCTION in *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Removing Registered JAR or ZIP Files

To remove an archive file (specifically, a JAR or ZIP file) that was previously registered using the SQLJ.INSTALL\_JAR external stored procedure, use the SQLJ.REMOVE\_JAR external stored procedure. If successful, REMOVE\_JAR removes a JAR or ZIP file installation and its classes from the database.

### Before You Begin

Take the following steps before you remove a previously registered archive file for an external routine:

1. Verify you have the EXECUTE privilege on the SQLJ.REMOVE\_JAR external stored procedure.
2. If the default database is not the same as the default database with which the archive file was registered, use the DATABASE statement to change the default database.
3. Verify you have the DROP PROCEDURE or DROP FUNCTION privilege on any of the following objects:
  - The default database
  - The object you are removing from the default database, identified by the SQL identifier that was passed to SQLJ.INSTALL\_JAR

## SQLJ.REMOVE\_JAR External Stored Procedure

### Syntax

```
REPLACE PROCEDURE SQLJ.REMOVE_JAR
  (IN jarname VARCHAR(128) CHARACTER SET UNICODE,
   IN undeploy INTEGER )
```

```
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SL!xsp!lib';
```

## Syntax Elements

### *jarname*

SQL identifier that was passed to SQLJ.INSTALL\_JAR, enclosed in apostrophes.

### *undeploy*

Integer value reserved for future use.

For now, the value of *undeploy* must be zero.

## Example: SQLJ.REMOVE\_JAR External Stored Procedure

Consider a JAR file that was registered with the JXSP database using the following statements:

```
DATABASE JXSP;
CALL SQLJ.INSTALL_JAR('SJ!java_xsp/accounts.jar', 'Accounts_JAR', 0);
```

The following statements remove the registered JAR file:

```
DATABASE JXSP;
CALL SQLJ.REMOVE_JAR('Accounts_JAR', 0);
```

## Replacing Registered JAR or ZIP Files

To replace an archive file (specifically, a JAR or ZIP file) that was previously registered using the external stored procedure SQLJ.INSTALL\_JAR, use the external stored procedure SQLJ.REPLACE\_JAR.

The replacement archive file must contain a class file for every class file in the original archive file that has been referenced in the EXTERNAL NAME clause of a CREATE/REPLACE PROCEDURE or CREATE/REPLACE FUNCTION statement. Class files that are in the original archive file but are not referenced by an external routine can be removed from the replacement archive file. The replacement archive file can contain new class files.

If the definition of a replacement class does not match the original definition, then executing the external routine might produce unpredictable results. If the parameter list of a method that implements a Java external routine uses data types that do not match the data types of the original method, then an error is returned indicating that no matching Java method was found.

Any currently executing SQL statements that use the archive file continue to use the original JAR or ZIP file until execution is complete.

## Before You Begin

Take the following steps before you replace a previously registered JAR or ZIP file for an external routine:

1. Verify you have the EXECUTE privilege on the SQLJ.REPLACE\_JAR external stored procedure.
2. If the default database is not the same as the default database with which the archive file was registered, use the DATABASE statement to change the default database.
3. Verify you have the DROP PROCEDURE or DROP FUNCTION privilege on any of the following objects:
  - The default database
  - The object that you are replacing within the default database, identified by the SQL identifier that was passed to SQLJ.INSTALL\_JAR

## SQLJ.REPLACE\_JAR External Stored Procedure

### Syntax

```
REPLACE PROCEDURE SQLJ.REPLACE_JAR
  (IN locspec VARCHAR(1000),
   IN jarname VARCHAR(128) CHARACTER SET UNICODE)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SL!xsplib';
```

### Syntax Elements

#### *locspec*

Location of the archive file that replaces the previously registered archive file.

The first two characters of the *locspec* string specify whether the replacement archive file is on the client or on the database server. If the first two characters are...

- CJ, then the .jar or .zip file is on the client.
- SJ, then the .jar or .zip file is on the database server.

The third character of the *locspec* string is a delimiter that you choose to separate the first two characters from the remaining characters in the *locspec* string.

The remaining characters in the *locspec* string specify the file path.

If the JAR or ZIP file is on the...

- client, then the file path is a client-interpreted path that specifies the location and name of the file, including the .jar or .zip file extension.
- database server, then the file path can be a full or relative path that specifies the location and name of the file, including the .jar or .zip file extension.

If the file path is relative, the full path to the archive file is formed by appending the relative path to the default path for source code on the server.

To determine the default path for source code, use the `-o` option of the *cufconfig* utility and find the setting for the `SourceDirectoryPath` field:

```
cufconfig -o
```

For more information on *cufconfig*, see *Teradata Vantage™ - Database Utilities*, B035-1102.

### ***jarname***

SQL identifier that was passed to `SQLJ.INSTALL_JAR`, enclosed in apostrophes.

## Redistributing Registered JAR or ZIP Files

You might need to redistribute a previously registered archive file (specifically, a JAR or ZIP file) to all nodes of a database system during copy, migrate, or system restore operations.

To redistribute a previously registered archive file to all nodes of a database system, use the `SQLJ.REDISTRIBUTE_JAR` external stored procedure. The archive file to be redistributed must have been registered using the `SQLJ.INSTALL_JAR` external stored procedure.

### **Before You Begin**

Take the following steps before you redistribute a previously registered archive file for an external routine:

1. Verify you have the `EXECUTE` privilege on the `SQLJ.REDISTRIBUTE_JAR` external stored procedure.
2. If the default database is not the same as the default database with which the archive file was registered, use the `DATABASE` statement to change the default database.
3. Verify you have the `DROP PROCEDURE` or `DROP FUNCTION` privilege on any of the following objects:
  - The default database
  - The object that you are redistributing, identified by the SQL identifier that was passed to `SQLJ.INSTALL_JAR`

## SQLJ.REDISTRIBUTE\_JAR External Stored Procedure

### **Syntax**

```
REPLACE PROCEDURE SQLJ.REDISTRIBUTE_JAR
  (IN jarname VARCHAR(128) CHARACTER SET UNICODE)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SL!xsp1ib';
```



## Syntax Elements

### *jarname*

SQL identifier that was passed to SQLJ.INSTALL\_JAR, enclosed in apostrophes.

## Example: SQLJ.REDISTRIBUTE\_JAR External Stored Procedure

Consider a JAR file that was registered with the JXSP database using the following statements:

```
DATABASE JXSP;
CALL SQLJ.INSTALL_JAR('SJ!java_xsp/accounts.jar', 'Accounts_JAR', 0);
```

The following statements redistribute the registered JAR file:

```
DATABASE JXSP;
CALL SQLJ.REDISTRIBUTE_JAR('Accounts_JAR');
```

## Altering the Java Path of Registered JAR or ZIP Files

To specify that the search path of classes for a registered JAR or ZIP file include classes from other JAR or ZIP files registered with the same database, use the SQLJ.ALTER\_JAVA\_PATH external stored procedure.

### Before You Begin

Take the following steps before you alter the Java path of a previously registered JAR or ZIP file for an external routine:

1. Verify you have the EXECUTE privilege on the SQLJ.ALTER\_JAVA\_PATH external stored procedure.
2. If the default database is not the same as the default database with which the JAR or ZIP file was registered, use the DATABASE statement to change the default database.
3. Verify you have the DROP PROCEDURE or DROP FUNCTION privilege on any of the following objects:
  - The default database
  - The object within the default database, identified by the SQL identifier that was passed to SQLJ.INSTALL\_JAR

## SQLJ.ALTER\_JAVA\_PATH External Stored Procedure

### Syntax

```
REPLACE PROCEDURE SQLJ.ALTER_JAVA_PATH (
  IN jarname VARCHAR(128) CHARACTER SET UNICODE,
  IN path VARCHAR(1000)
)
LANGUAGE C
```

```
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SL!xsp1ib';
```

***path***

```
(referenced_class, resolution_archivename)
```

**Syntax Elements*****jarname***

SQL identifier that was passed to SQLJ.INSTALL\_JAR, enclosed in apostrophes.

***path***

Registered JAR or ZIP files to search when resolving class references for the archive file specified by *jarname*.

Specify an empty string for *path* to remove all classes and JAR or ZIP file identifiers from the search path for the classes in the JAR or ZIP file specified by *jarname*.

***referenced\_class***

Specifies an asterisk ( \* ), meaning that the search path is to include all classes from the JAR or ZIP file identified by *resolution\_archivename*.

***resolution\_archivename***

Specifies the SQL identifier of an archive file that was registered with the same database as the JAR or ZIP file specified by *jarname*.

**Example: SQLJ.ALTER\_JAVA\_PATH External Stored Procedure**

The following statements specify that the search path of classes in the JAR file that has an SQL identifier called Accounts\_JAR include classes from the JAR file that has an SQL identifier called Orders\_JAR and classes from the JAR file that has an SQL identifier called Sales\_JAR:

```
DATABASE JXSP;
CALL SQLJ.ALTER_JAVA_PATH('Accounts_JAR',
                          '(*,Orders_JAR) (*, Sales_JAR)');
```

You can use any of the following statements to remove all classes from the search path of classes in the JAR file that has an SQL identifier called Accounts\_JAR:

```
CALL SQLJ.ALTER_JAVA_PATH('Accounts_JAR', '');
CALL SQLJ.ALTER_JAVA_PATH('Accounts_JAR', '');
```

```
CALL SQLJ.ALTER_JAVA_PATH('Accounts_JAR', '()');
CALL SQLJ.ALTER_JAVA_PATH('Accounts_JAR', '( )');
CALL SQLJ.ALTER_JAVA_PATH('Accounts_JAR', '() ');
```

## Distributing Packages

Teradata provides the following tools for you to use to distribute packages to all nodes of an MPP system:

- A system stored procedure called `installsp`, located in the SYSLIB database
- A Microsoft Windows, Call-Level Interface Version 2 (CLv2) based executable called `lobteq` from which you call the `installsp` stored procedure

Additionally, Teradata provides the following system tables in the SYSLIB database for storing information for versioning, backing up, and restoring packages:

- `dem`
- `demddl`
- `dempart`

Vantage maintains the system tables and updates the information when you back up or restore packages. For details, see [Backing Up and Restoring Packages \[Deprecated\]](#).

### Before You Begin

Before you can distribute any packages, download `lobteq` to your Windows client system and verify that the SYSLIB system database contains the `installsp` stored procedure and `dem`, `demddl`, and `dempart` tables.

1. Use the following SQL statement to view the objects in the SYSLIB system database:

```
HELP DATABASE SYSLIB;
```

2. If the information that `HELP DATABASE` returns does not include the `installsp` stored procedure or the `dem`, `demddl`, `dempart` tables, execute the `DIPDEM` script using the Database Initialization Program (DIP) utility.

For more information on DIP, see *Teradata Vantage™ - Database Utilities*, B035-1102.

3. Verify the following privileges:
  - You must have `EXECUTE PROCEDURE` on the `SYSLIB.installsp` stored procedure.
  - The SYSLIB database must have `EXECUTE FUNCTION` on the `SYSLIB.installpkg` UDF.
4. Download `lobteq` to your Windows client system:
  - a. Log on to your Windows client system using an Administrator account.
  - b. Verify Teradata Tools and Utilities is installed and that all the product dependencies for BTEQ are satisfied.

For more information on BTEQ product dependencies, see *Teradata® Tools and Utilities for Microsoft Windows Installation Guide* (B035-2407).

- c. Go to [Teradata Downloads](#) and download the UDF Packaging self extractable file to a directory of your choice.
- d. Extract the lobteq executable from the self extractable file.

## installsp Stored Procedure

Use the installsp stored procedure to distribute a package to all nodes of an MPP system.

### Syntax

```
REPLACE PROCEDURE installsp
  (IN name      VARCHAR(250) CHARACTER SET LATIN,
   IN version   VARCHAR(250) CHARACTER SET LATIN,
   IN filename  VARCHAR(250) CHARACTER SET LATIN,
   IN path      VARCHAR(250) CHARACTER SET LATIN,
   IN source    BLOB(200000),
   IN operation VARCHAR(250) CHARACTER SET LATIN,
   OUT pform    VARCHAR(20)  CHARACTER SET LATIN,
   OUT cfspath  VARCHAR(200) CHARACTER SET LATIN
  )
```

### Syntax Elements

#### *name*

Name for the package.

The name that you use here to distribute the package is the same name that you specify later in the CREATE/REPLACE statement that you use to install the package.

#### *version*

Version you assign to the package.

For example, when you initially distribute a package, you might want to assign a version of '1.0'. When you update the package and redistribute it, you might want to assign a version of '2.0' or '1.1', depending on the changes to the package.

#### *filename*

Name to use for the server-side destination file.

The filename you specify here to distribute the package is the same filename that you specify later in the CREATE/REPLACE statement to install the package.

***path***

Optional path to use for the server-side destination file. The path you specify is relative to the following fixed path:

*teradata\_installation\_path*/Teradata/dem

where *teradata\_installation\_path* is the path of the Teradata installation. To get the value of *teradata\_installation\_path*, enter the following on the command line:

```
pdepath -I
```

The full path to the server-side destination file is formed by appending the relative path specified by *path* to the fixed path. If you omit the path argument, the full path is the same as the fixed path.

The `installsp` stored procedure returns the full path in the *cfgpath* OUT argument.

***source***

USING variable name preceded by a COLON character.

When you call `installsp` to distribute a package, the CALL statement that you use must specify a USING row descriptor. During processing, *installsp* replaces *source* with the contents of the package.

***operation***

the operation to perform on the package. The valid values are listed below. They are case insensitive.

- 'CREATE', which specifies to copy the contents of the package specified by *source* to all nodes in the location specified by *path* with the name specified by *filename*. If the file already exists, do not overwrite it.
- 'REPLACE', which specifies to copy the contents of the package specified by *source* to all nodes in the location specified by *path* with the name specified by *filename*. Overwrite the target file if it exists.
- 'DROP', which specifies to delete the file named *filename* in the location specified by *path*.
- 'CHECK', which specifies to not perform an operation on the package, but simply return the target platform type in the *pform* argument.

***pform***

Target platform type, for example LINUX64.

***cfgpath***

Full path on the server where the package has been distributed, formed by appending the relative path specified by *path* to the fixed path for the target platform.

Later, when you use an appropriate CREATE/REPLACE statement to install the package, use the return value of *cfgpath* in the EXTERNAL NAME clause to specify the full path to the server-side destination file.

## Usage Notes

### Procedure for Distributing Packages

Here is a synopsis of the steps you take to distribute a package to all nodes of an MPP system. For a complete example, see .

1. On the target server, modify the access privileges of the directory for the server-side destination file to allow access by the user 'tdatuser'. (The path to the server-side destination file uses the path argument of the *installsp* stored procedure.)

Alternatively, to distribute a package under the authorization of an operating system user other than 'tdatuser', follow these steps:

- a. Use the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement to create a context that identifies an operating system user to use for package distribution.
- b. Use the following SQL statement to get the definition of the SYSLIB.INSTALLPKG UDF (The *installsp* stored procedure calls the *installpkg* UDF.):

```
SHOW FUNCTION SYSLIB.INSTALLPKG;
```

- c. Use the REPLACE FUNCTION statement to modify the SYSLIB.INSTALLPKG definition to include the EXTERNAL SECURITY clause, associating execution of the UDF with the context created by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement.
- d. On the target server, modify the access privileges of the directory for the server-side destination file to allow access by the user established by the CREATE AUTHORIZATION or REPLACE AUTHORIZATION statement. (The path to the server-side destination file uses the path argument of the *installsp* stored procedure.)

For details on CREATE AUTHORIZATION, REPLACE AUTHORIZATION, REPLACE FUNCTION, and the EXTERNAL SECURITY clause, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

2. Start *lobteq* on your Windows client system.

The interface and commands are similar to BTEQ.

3. Log on to the target database.

Use the same LOGON command that you use in BTEQ. For details on the BTEQ LOGON command, see *Basic Teradata® Query Reference*, B035-2414.

4. Use the *lobteq .using* command to open the package and load it into a data buffer for the USING modifier of the next CALL statement.

The `.using` command supports the following syntax for opening a package:

```
.using 'filename'xfile
```

where *filename* is the filename, including the path, of the package.

5. Call the `SYSLIB.installsp` stored procedure.

Use a `USING` modifier with the `CALL` statement to pass the package as a LOB in deferred mode to the server.

6. Exit `lobteq`.
7. [Optional] On the server, modify the access privileges of the directory for the server-side destination file to remove access by the operating system user used for package distribution.

## Installing a Distributed Package

After you distribute your package, you can use the appropriate `CREATE/REPLACE` statement to install the package on the server. For example, after you distribute a UDF package, use the `CREATE FUNCTION` or `REPLACE FUNCTION` statement to install the UDF package on the server.

Teradata provides a stored procedure that you can use to save the `CREATE/REPLACE` statement with the package and version information in the `dem`, `demddl`, and `dempart` tables in the `SYSLIB` database. For details, see [Procedure for Backing Up a Package](#).

## Example: Distributing a Package

Here is an example of how to distribute version 1.0 of a package called `libcstd2.so` to all nodes. In this example, the target server directory is `/Teradata/dem/udfs` and there is a `.so` file on the Windows client:

```
LobTeq -- Enter your DBC/SQL request or LobTeq command:
.using 'libcstd2.so'xfile

*** .using accepted

LobTeq -- Enter your DBC/SQL request or LobTeq command:
USING (a BLOB AS DEFERRED)
CALL SYSLIB.installsp('cstdlib2', '1.0', 'libcstd2.so', 'udfs',
    :a, 'CREATE', pform, cfgpath);

Sending LOB data, chunk 1
*** Procedure has been executed.

Linux64 /Teradata/dem/udfs/
```

Here is an example of a CREATE FUNCTION statement that creates the function from the installed package.

```
CREATE FUNCTION SYSLIB.cSTD_DEV(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SP!/Teradata/dem/udfs/libcstd2.so';
```

Note that the filename and path specified in the EXTERNAL NAME clause are the same as the *filename* IN argument and *cfgpath* OUT argument in the call to `installsp`.

## Related Information

FOR information on ...	SEE ...
the commands that <code>lobteq</code> supports	the information that <code>lobteq</code> provides when you use the <code>.</code> help command.
installing a UDF package using the CREATE FUNCTION statement	<i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.
backing up or restoring a package	<a href="#">Backing Up and Restoring Packages [Deprecated]</a> .

## Backing Up and Restoring Packages [Deprecated]

Teradata provides the following tools for you to use to back up and restore packages and package components, such as source and include files:

- System stored procedures called *savepkg* and *loadpkg*, located in the SYSLIB database
- A Microsoft Windows, CLlV2-based executable called *lobteq* from which you call the *savepkg* and *loadpkg* stored procedures

Additionally, Teradata provides the following system tables in the SYSLIB database for storing information for versioning, backing up, and restoring packages:

- `dem`
- `demddl`
- `dempart`

The database maintains the system tables and updates the contents when you back up or restore packages or package components.



## Before You Begin

Before you can back up or restore any packages or package components, verify that *lobteq* is available for you to use on your Windows client system and that the SYSLIB system database contains the *savepkg* and *loadpkg* stored procedures and *dem*, *demddl*, and *dempart* tables.

For details on how to do this, see [Before You Begin](#).

## savepkg Stored Procedure

Use the *savepkg* stored procedure to back up a specific version of a package and the associated DDL statements that install the package, storing the information in the *demddl* and *dempart* system tables.

### Syntax

```
REPLACE PROCEDURE savepkg
  (IN  U_Name      VARCHAR(30),
   IN  U_Version   VARCHAR(30),
   IN  U_Database  VARCHAR(30),
   IN  U_DemT      VARCHAR(30),
   IN  U_Create    CLOB,
   IN  U_DemFN     VARCHAR(30),
   IN  U_Content   BLOB,
   OUT out1        VARCHAR(30)
  )
```

### Syntax Elements

#### *U\_Name*

Name of the package.

Vantage stores the name of the package in the *demddl* and *dempart* system tables in the SYSLIB database.

#### *U\_Version*

Version of the package.

Vantage stores the version number that you assign to the package in the *demddl* and *dempart* system tables in the SYSLIB database.

#### *U\_Database*

Database in which the package is installed.

Vantage stores the name of the database in which you installed the package in the demddl and dempart system tables in the SYSLIB database.

### ***U\_DemT***

Value that you assign to describe the type of package or package component.

Vantage stores the value that you provide in the demddl system table in the SYSLIB database.

For example, if you are using savepkg to back up a UDF package, you might want to use the value 'UP'. If you are using savepkg to back up UDF package components, you might want to use 'US' for UDF source, 'UI' for UDF include files, and 'UO' for UDF objects.

### ***U\_Create***

DDL statement that installs the package on the server.

If the DDL statement is in a file, you can use a USING variable name preceded by a COLON character. During processing, savepkg replaces *U\_Create* with the contents of the file.

If the DDL statement specifies an EXTERNAL NAME clause with multiple parts, for example a C language source file and a C language include file, you must call savepkg for each part, but you only need to pass the DDL statement in the first call. In each subsequent call, you can pass 'NULL' for the *U\_Create* argument.

Vantage stores the statement in the demddl system table in the SYSLIB database.

### ***U\_DemFN***

Filename of the package.

### ***U\_Content***

USING variable name preceded by a COLON character.

When you call savepkg to back up a package, the CALL statement that you use must specify a USING row descriptor. During processing, savepkg replaces *U\_Content* with the contents of the package.

### ***out1***

Return status.

## **loadpkg Stored Procedure**

Use the loadpkg stored procedure to retrieve scripts that you can use to restore a specific version of a package.

To generate the scripts, `loadpkg` uses the information that was stored in the `demddl` and `dempart` system tables by a previous call to `savepkg`.

## Syntax

```
REPLACE PROCEDURE loadpkg
  (IN  U_Name      VARCHAR(30),
   IN  U_Version   VARCHAR(30),
   IN  U_Database  VARCHAR(30),
   IN  U_Machine   VARCHAR(30),
   IN  U_InfoType  VARCHAR(30),
   OUT script      CLOB
  )
```

## Syntax Elements

### *U\_Name*

Name of the package.

The package name that you specify here must be the same as a package name that you passed in a previous call to `savepkg`.

### *U\_Version*

the version of the package.

The version that you specify here must be the same as a version that you passed in a previous call to `savepkg`.

### *U\_Database*

the name of the database in which the package is installed.

The database name that you specify here must be the same as a database name that you passed in a previous call to `savepkg`.

### *U\_Machine*

the identifier associated with the Teradata server.

The *U\_Machine* identifier is the same identifier you use to log on to the Teradata server.

### *U\_InfoType*

the type of contents that `loadpkg` returns in the *script* OUT argument. The valid values are:

- 'DDL', which specifies to return the DDL statements required to load the package into the database.

- 'PART', which specifies to return the lobteq commands to retrieve all of the files associated with the package from the SYSLIB tables.

**script**

DDL statements or lobteq commands, where the contents depend on the value of the *U\_InfoType* IN argument.

## Procedure for Backing Up a Package

Here is a synopsis of the steps you take to back up a specific version of a package and the associated CREATE/REPLACE statement that installs the package on the server. For a complete example, see [Example: Backing Up a Package](#).

1. Save the CREATE/REPLACE statement that installs the package on the server in a file.
2. Start lobteq on your Windows client system.

The interface and commands are similar to BTEQ.

3. Log on to the target database.

Use the same LOGON command that you use in BTEQ. For details on the BTEQ LOGON command, see *Basic Teradata® Query Reference*, B035-2414.

4. Use the lobteq .using command to open the package and the file that you created in Step 1 and load them into data buffers for the USING modifier of the next CALL statement.

The .using command supports the following syntax for opening files:

```
.using 'filename'.xfile [... 'filename'.xfile]
```

where *filename* is the name of the file.

5. Call the savepkg stored procedure.

Use a USING modifier with the CALL statement to pass the contents of the two files as LOBs in deferred mode to the server.

6. Exit lobteq.

The savepkg stored procedure backs up the specific version of your package and its associated CREATE/REPLACE statement in the SYSLIB system tables, which are FALLBACK protected. You can then add the SYSLIB system tables to the list of database tables that you back up as part of your regular archiving activities.

## Example: Backing Up a Package

Here is an example of how to back up version 1.0 of a package called libcstd2.so and its associated CREATE statement that installs the package on the Teradata server.

```
lobteq << EOF

.logon slugger/adm455,sn6Y24

.using 'cstdlib2_DDL.txt'xfile 'libcstd2.so'xfile
USING (a CLOB AS DEFERRED, b CLOB AS DEFERRED)
CALL SYSLIB.savepkg('cstdlib2','1.0','SYSLIB','UP', :a,
    'libcstd2.so', :b, out1);

EOF
```

The CREATE statement in the cstdlib2\_DDL.txt file looks something like this:

```
CREATE FUNCTION SYSLIB.cSTD_DEV(x FLOAT)
RETURNS FLOAT
CLASS AGGREGATE
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'SP!/Teradata/dem/udfs/libcstd2.so';
```

## Procedure for Restoring a Package

Here is a synopsis of the steps you take to restore a package that has previously been saved in the demddl and dempart system tables using the savepkg stored procedure.

1. Start lobteq on your Windows client system.

The interface and commands are similar to BTEQ.

2. Log on to the target database.

Use the same LOGON command that you use in BTEQ. For details on the BTEQ LOGON command, see *Basic Teradata® Query Reference*, B035-2414.

3. Use the lobteq .exportfiles command to open a file to store the script that the next call to loadpkg returns.

The syntax for the .exportfiles command is as follows:

```
.exportfiles filename
```

4. Call the loadpkg stored procedure, setting the value of the *U\_InfoType* argument to 'DDL'.

To handle the *script* OUT argument, which is passed by locator, precede the CALL statement with the lobteq .cursor command.

5. Use a SELECT statement and USING modifier that defines a CLOB AS LOCATOR parameter to export the script data to the file that was opened with the .exportfiles command.
6. Repeat steps 3 to 5, setting the *U\_InfoType* argument of the loadpkg stored procedure to 'PART'.
7. Exit lobteq.
8. Run the second script that was retrieved through lobteq to redistribute the package.
9. Run the first script that was retrieved through BTEQ to submit the DDL statements to reinstall the package.

### Example: Restoring a Package

Here is an example of how to restore version 1.0 of a package called libcstd2.so.

First, call the loadpkg stored procedure twice: once to retrieve a script of the DDL statement that installs the package onto the server and again to retrieve a script of the lobteq commands to retrieve the package from the SYSLIB tables.

```
lobteq << EOF
.logon slugger/adm455,sn6Y24

.exportfiles cstdlib2_DDL.txt
.cursor
CALL SYSLIB.loadpkg('cstdlib2','1.0','SYSLIB','slugger','DDL',
script);
USING (a CLOB AS LOCATOR)
SELECT :a;
.exportfiles cstdlib2_lobload.txt
.cursor
CALL loadpkg('cstdlib2','1.0','SYSLIB','slugger','PART',script);
USING (a CLOB AS LOCATOR)
SELECT :a;

EOF
```

Next, redistribute Version 1.0 of the libcstd2.so package by executing the cstdlib2\_lobload.txt script:

```
lobteq < cstdlib2_lobload.txt
```

Finally, submit the DDL statements to reinstall Version 1.0 of the libcstd2.so package by executing the xmlib\_DDL.txt script:

```
bteq < cstdlib2_DDL.txt
```

## SYSLIB Tables

The following table identifies the column names and contents of the dem, demddl, and dempart FALLBACK protected tables in the SYSLIB database.

Table Name and Description	Column and Type	Column Description
SYSLIB.dem Stores the currently installed version of packages. The <i>loadpkg</i> stored procedure populates this table when you restore a specific version of a package to a target database.	name VARCHAR(30)	Name of the package.
	version VARCHAR(30)	Version of the currently installed package.
	dbase VARCHAR(30)	Database in which the package is installed.
	itime TIMESTAMP(6)	Date of the installation.
SYSLIB.demddl Stores the CREATE/REPLACE DDL information of all packages.	name VARCHAR(30)	Name of the package associated with the DDL statements. This is the value of the <i>U_Name</i> argument in a call to <i>savepkg</i> stored procedure.
	version VARCHAR(30)	Version of the package associated with the DDL statements. This is the value of the <i>U_Version</i> argument in a call to <i>savepkg</i> stored procedure.
	dbase VARCHAR(30)	Target database of the package installation. This is the value of the <i>U_Database</i> argument in a call to <i>savepkg</i> stored procedure.
	ddlcontent CLOB(2097088000)	CREATE/REPLACE statement for installing the package identified by the name and version columns. This is the value of the <i>U_Create</i> argument in a call to <i>savepkg</i> stored procedure.
	demtype VARCHAR(5)	Type of package or package component that the CREATE/REPLACE statement creates. This is the value of the <i>U_DemT</i> argument in a call to <i>savepkg</i> stored procedure.
	itime TIMESTAMP(6)	Date when the DDL information was saved.
SYSLIB.dempart Stores each part of a package.	name VARCHAR(30)	Name of the package associated with the part.

Table Name and Description	Column and Type	Column Description
For example, a part could be in source, object, or library format.		This is the value of the <i>U_Name</i> argument in a call to <i>savepkg</i> stored procedure.
	version VARCHAR(30)	Version of the package associated with the part. This is the value of the <i>U_Version</i> argument in a call to <i>savepkg</i> stored procedure.
	dbase VARCHAR(30)	Target database of the package installation. This is the value of the <i>U_Database</i> argument in a call to <i>savepkg</i> stored procedure.
	partfn VARCHAR(30)	Filename of the package. This is the value of the <i>U_DemFN</i> argument in a call to <i>savepkg</i> stored procedure.
	partcontent BLOB(2097088000)	Package content. This is the value of the <i>U_Content</i> argument in a call to <i>savepkg</i> stored procedure.

## SYSLIB Space Considerations

The DIPDEM script that you execute using DIP initially allocates 15000000 bytes of permanent space to the SYSLIB database.

Each call you make to *savepkg* and *loadpkg* to store package information in the dem, demddl, and dempart tables requires space in the SYSLIB database.

As the package information that you store in the dem, demddl, and dempart tables grows, you need to increase the permanent space for SYSUDTLIB accordingly.

## Related Information

FOR information on ...	SEE ...
the commands that <i>lobteq</i> supports	the information that <i>lobteq</i> provides when you use the . help command.
installing a UDF package using the CREATE FUNCTION statement	<i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.
distributing a package	<a href="#">Distributing Packages</a> .



# SQL Data Type Mapping

The following sections describe the C and Java data types that you can use for the arguments and results of UDFs, UDMs, and external stored procedures and how they map to SQL data types.

## C Data Types

The following pages show the SQL data types and the corresponding C data types that you can use as parameters and return types of UDFs, UDMs, or external stored procedures where the source code is C or C++.

For exact definitions, see the `sqltypes_td.h` header file.

### *array\_name*

#### C Data Type Definition

```
typedef int ARRAY_HANDLE;
```

#### Usage

Use the `ARRAY_HANDLE` data type when passing an `ARRAY` argument or returning an `ARRAY` type result.

Here is an example using an `ARRAY` parameter in a UDF definition and `ARRAY_HANDLE` in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A phonenumbers_ary) RETURNS phonenumbers_ary ...;</pre>	<pre>void f1( ARRAY_HANDLE *a,         ARRAY_HANDLE *result,         ... ) { ... }</pre>

## BIGINT

#### C Data Type Definition

```
typedef long long BIGINT;
```

#### Usage

Here is an example using `BIGINT` in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1(   A BIGINT ) RETURNS BIGINT ...;</pre>	<pre>void f1( BIGINT *a,         BIGINT *result,         ... ) { ... }</pre>

## BINARY LARGE OBJECT / BLOB

### C Data Type Definition

```
typedef int LOB_LOCATOR;
typedef int LOB_RESULT_LOCATOR;
```

### Usage

BLOBs are passed as locators only, and not automatically loaded into memory. BLOB argument and return types must specify AS LOCATOR in the CREATE FUNCTION statement.

Here is an example using BLOB AS LOCATOR in a UDF definition and LOB\_LOCATOR and LOB\_RESULT\_LOCATOR in the C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A BLOB AS LOCATOR) RETURNS BLOB AS LOCATOR ...;</pre>	<pre>void f1( LOB_LOCATOR      *a,         LOB_RESULT_LOCATOR *result,         ... ) { ... }</pre>

## BYTE

### C Data Type Definition

```
typedef unsigned char BYTE;
```

### Usage

Fixed length byte data that requires padding uses a binary zero as its pad character.

If the function result type is fixed length byte data, then the result argument points to a data area with the size set to the maximum defined by the RETURNS clause in the CREATE FUNCTION statement.

Here is an example using BYTE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A BYTE(30) ) RETURNS BYTE(12) ...;</pre>	<pre>void f1( BYTE a[30],         BYTE *result,         ... ) { ... }</pre>

## BYTEINT

### C Data Type Definition

```
typedef signed char BYTEINT;
```

### Usage

Here is an example using BYTEINT in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A BYTEINT ) RETURNS BYTEINT ...;</pre>	<pre>void f1( BYTEINT *a,         BYTEINT *result,         ... ) { ... }</pre>

## CHARACTER / CHAR

### C Data Type Definition

```
typedef unsigned char CHARACTER;
typedef Latin_Text CHARACTER_LATIN;
typedef Kanjisjis_Text CHARACTER_KANJISJIS;
typedef Kanji1_Text CHARACTER_KANJI1;
typedef Unicode_Text CHARACTER_UNICODE;
```

### Usage

Character data types are passed using the standard null-terminated C string. Because of this, the length of the C string is one more than the length of the SQL string.

If you want to allow a binary zero character in the middle of a UDF input parameter string, use the CHAR data type because your C code can more easily handle embedded binary zero characters in fixed length strings. Similarly, if you want to allow a binary zero character in the middle of a UDF result string, use the CHAR data type because a binary zero character in the middle of a VARCHAR result string will end the string.

If the function defines a fixed length character input parameter, the number of characters in the input string always matches the maximum defined by the CREATE FUNCTION statement. If the function is called with a string that has fewer characters than the maximum, the input string is padded on the right before it is passed to the function.

Fixed length character data that requires padding uses the space character for the specified character set.

If the function result type is fixed length character data, then the result argument points to a data area with the size set to the maximum defined by the CREATE FUNCTION statement, plus one for the null string terminator. The number of characters in the result string must match the maximum defined by the CREATE FUNCTION statement. If the returned data is less than the maximum, you must add padding to the result string.

If the character set is UNICODE, use the CHARACTER\_UNICODE data type. For character sets other than UNICODE, you can use the CHARACTER data type.

This example uses CHAR in a UDF definition and CHARACTER\_LATIN in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A CHAR(6) CHARACTER SET LATIN) RETURNS CHAR CHARACTER SET LATIN ...;</pre>	<pre>void f1(Character_Latin a[7],         Character_Latin *result,         ... ) { ... }</pre>

## Restrictions

Unicode strings use two bytes per character, including the null string terminator. The C library wide characters (wchar\_t) are four bytes; therefore, you cannot use the C library wide character routines to manipulate Unicode strings.

## CHARACTER CHARACTER SET GRAPHIC

### C Data Type Definition

```
typedef unsigned short GRAPHIC;
```

### Usage

Fixed length graphic data that requires padding uses the ideographic space (U+3000).

If the function result type is fixed length graphic data, then the result argument points to a data area with the size set to the maximum defined by the RETURNS clause in the CREATE FUNCTION statement.

Here is an example using CHARACTER(n) CHARACTER SET GRAPHIC in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A CHARACTER(30) CHARACTER SET GRAPHIC) RETURNS CHARACTER(12) CHARACTER SET GRAPHIC ...;</pre>	<pre>void f1( GRAPHIC *a,         GRAPHIC *result,         ... ) { ... }</pre>

## CHARACTER VARYING / LONG VARCHAR / VARCHAR

### C Data Type Definition

```
typedef Latin_Text VARCHAR_LATIN;
typedef Kanjisjis_Text VARCHAR_KANJISJIS;
typedef Kanji1_Text VARCHAR_KANJI1;
typedef Unicode_Text VARCHAR_UNICODE;
```

### Usage

Character data types are passed using the standard null-terminated C string. The length of the C string is one more than the length of the SQL string.

If you want to allow a binary zero character in the middle of a UDF input parameter string, use the CHAR data type instead of VARCHAR because your C code can more easily handle embedded binary zero characters in fixed length strings. Similarly, if you want to allow a binary zero character in the middle of a UDF result string, use the CHAR data type because a binary zero character in the middle of a VARCHAR result string will end the string.

If the function result type is a varying length character type, then the result argument points to a data area with the size set to the maximum defined by the CREATE FUNCTION statement, plus one for the null string terminator. The function code must set the correct length of the result by using a null string terminator to end the string.

For LONG VARCHAR, the length is 64000 single byte characters or 32000 double byte characters.

This example uses VARCHAR in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A VARCHAR(30) ) RETURNS VARCHAR(12) ...;</pre>	<pre>void f1( VARCHAR_LATIN *a,         VARCHAR_LATIN *result,         ... ) { ... }</pre>

## Restrictions

Unicode strings use two bytes per character, including the null string terminator. The C library wide characters (wchar\_t) are four bytes; therefore, you cannot use the C library wide character routines to manipulate Unicode strings.

## CHARACTER LARGE OBJECT / CLOB

### C Data Type Definition

```
typedef long LOB_LOCATOR;
typedef long LOB_RESULT_LOCATOR;
```

### Usage

CLOBs are passed as locators only, and not automatically loaded into memory. CLOB argument and return types must specify AS LOCATOR in the CREATE FUNCTION statement.

Here is an example using CLOB AS LOCATOR in a UDF definition and LOB\_LOCATOR and LOB\_RESULT\_LOCATOR in the C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A CLOB AS LOCATOR) RETURNS CLOB AS LOCATOR ...;</pre>	<pre>void f1( LOB_LOCATOR      *a,         LOB_RESULT_LOCATOR *result,         ... ) { ... }</pre>

## DATASET

### C Data Type Definition

```
typedef int DATASET_HANDLE;
```

### Usage

Use the DATASET\_HANDLE data type when passing a DATASET instance as an argument or returning a DATASET type result.

Here is an example using a DATASET parameter in a UDF definition and DATASET\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A DATASET(100) STORAGE FORMAT AVRO)</pre>	<pre>void f1( DATASET_HANDLE *a,         DATASET_HANDLE *result,</pre>

SQL Function Definition	Equivalent C Function Declaration
<pre>RETURNS DATASET(100) STORAGE FORMAT AVRO ...;</pre>	<pre>... ) { ... }</pre>

## DATE

### C Data Type Definition

```
typedef long int DATE;
```

### Usage

The date is represented using the following formula:

$((year-1900) \times 10000 + (month \times 100) + day)$

Here is an example using DATE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A DATE ) RETURNS DATE ...;</pre>	<pre>void f1( DATE *a,         DATE *result,         ... ) { ... }</pre>

## DECIMAL / NUMERIC

### C Data Type Definition

```
typedef signed char DECIMAL1;
typedef short      DECIMAL2;
typedef int        DECIMAL4;

typedef signed char NUMERIC1;
typedef short      NUMERIC2;
typedef int        NUMERIC4;

typedef struct
{
  unsigned int low;
  int          high;
} DECIMAL8, NUMERIC8;
```

```
typedef struct
{
    unsigned int int1;
    unsigned int int2;
    unsigned int int3;
    int         int4;
} DECIMAL16, NUMERIC16;
```

## Usage

The size of the SQL type determines which C type to use.

FOR DECIMAL(n,m) or NUMERIC(n,m), where ...	Use one of these C types ...
$1 \leq n \leq 2$	DECIMAL1 or NUMERIC1
$3 \leq n \leq 4$	DECIMAL2 or NUMERIC2
$5 \leq n \leq 9$	DECIMAL4 or NUMERIC4
$10 \leq n \leq 18$	DECIMAL8 or NUMERIC8
$18 < n$	DECIMAL16 or NUMERIC16

The size of the SQL type also determines the range of values for a DECIMAL input argument or return argument. For example, if a CREATE FUNCTION statement defines a DECIMAL(1,0) input argument, the range of values for the argument is -9 to 9. A value outside the valid range of values produces a numeric overflow error. For details on DECIMAL types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

For a DECIMAL type that can be represented as variable length, the best practice is to specify the DECIMAL so that it can handle the largest value that the C code is willing to deal with.

This example uses DECIMAL in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A DECIMAL(5,0) ) RETURNS DECIMAL(5,0) ...;</pre>	<pre>void f1( DECIMAL4 *a,         DECIMAL4 *result,         ... ) { ... }</pre>



## DOUBLE PRECISION / FLOAT / REAL

### C Data Type Definition

```
typedef double REAL;
typedef double DOUBLE_PRECISION;
typedef double FLOAT;
```

### Usage

Here is an example using FLOAT in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A FLOAT ) RETURNS FLOAT ...;</pre>	<pre>void f1( FLOAT *a,         FLOAT *result,         ... ) { ... }</pre>

## Geospatial: MBB, MBR, ST\_GEOMETRY

### C Data Type Definition

```
typedef int GEO_HANDLE;
```

### Usage

Use the GEO\_HANDLE data type when passing an MBB, MBR, or ST\_GEOMETRY type as an argument or returning an MBB, MBR, or ST\_GEOMETRY type result.

Here is an example using an ST\_GEOMETRY parameter in a UDF definition and GEO\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A ST_GEOMETRY(1000)) RETURNS ST_GEOMETRY(1000) ...;</pre>	<pre>void f1( GEO_HANDLE *a,         GEO_HANDLE *result,         ... ) { ... }</pre>

## INTEGER

### C Data Type Definition

```
typedef int INTEGER;
```

### Usage

This example uses INTEGER in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTEGER )   RETURNS INTEGER   ...;</pre>	<pre>void f1( INTEGER *a,         INTEGER *result,         ... ) {  ... }</pre>

## INTERVAL DAY

### C Data Type Definition

```
typedef SMALLINT INTERVAL_DAY;
```

### Usage

The range of values defined for the SQL INTERVAL DAY type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL DAY type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

This example uses INTERVAL DAY in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL DAY(2) )   RETURNS INTERVAL DAY   ...;</pre>	<pre>void f1( INTERVAL_DAY *a,         INTERVAL_DAY *result,         ... ) {  ... }</pre>

## INTERVAL DAY TO HOUR

### C Data Type Definition

```
typedef struct IntrvlDtoH
{
```

```

    SMALLINT day;
    SMALLINT hour;
} IntrvlDtoH;

```

## Usage

The range of values defined for the SQL INTERVAL DAY TO HOUR type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL DAY TO HOUR type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL DAY TO HOUR in a UDF definition and IntrvlDtoH in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre> CREATE FUNCTION F1 (   A INTERVAL DAY TO HOUR ) RETURNS INTERVAL DAY TO HOUR ...; </pre>	<pre> void f1( IntrvlDtoH *a,         IntrvlDtoH *result,         ... ) { ... } </pre>

## INTERVAL DAY TO MINUTE

### C Data Type Definition

```

typedef struct IntrvlDtoM
{
    SMALLINT day;
    SMALLINT hour;
    SMALLINT minute;
    short pad;
} IntrvlDtoM;

```

## Usage

The range of values defined for the SQL INTERVAL DAY TO MINUTE type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL DAY TO MINUTE type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL DAY TO MINUTE in a UDF definition and IntrvlDtoM in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre> CREATE FUNCTION F1 (   A INTERVAL DAY TO MINUTE ) </pre>	<pre> void f1( IntrvlDtoM *a,         IntrvlDtoM *result, </pre>

SQL Function Definition	Equivalent C Function Declaration
<pre>RETURNS INTERVAL DAY TO MINUTE ...;</pre>	<pre>... ) { ... }</pre>

## INTERVAL DAY TO SECOND

### C Data Type Definition

```
typedef struct IntrvlDtoS
{
    DECIMAL4 seconds;
    SMALLINT day;
    SMALLINT hour;
    SMALLINT minute;
} IntrvlDtoS;
```

### Usage

The first member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds. For details, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

The range of values defined for the SQL INTERVAL DAY TO SECOND type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL DAY TO SECOND type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL DAY TO SECOND in a UDF definition and IntrvlDtoS in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL DAY(4) TO SECOND(3)) RETURNS INTERVAL DAY TO SECOND ...;</pre>	<pre>void f1( IntrvlDtoS *a,         IntrvlDtoS *result,         ... ) { ... }</pre>

## INTERVAL HOUR

### C Data Type Definition

```
typedef SMALLINT HOUR;
```

## Usage

The range of values defined for the SQL INTERVAL HOUR type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL HOUR type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL HOUR in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL HOUR ) RETURNS INTERVAL HOUR ...;</pre>	<pre>void f1( HOUR *a,         HOUR *result,         ... ) { ... }</pre>

## INTERVAL HOUR TO MINUTE

### C Data Type Definition

```
typedef struct IntrvlHtoM
{
    SMALLINT hour;
    SMALLINT minute;
} IntrvlHtoM;
```

## Usage

The range of values defined for the SQL INTERVAL HOUR TO MINUTE type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL HOUR TO MINUTE type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL HOUR TO MINUTE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL HOUR(4) TO MINUTE ) RETURNS INTERVAL HOUR TO MINUTE ...;</pre>	<pre>void f1( IntrvlHtoM *a,         IntrvlHtoM *result,         ... ) { ... }</pre>

## INTERVAL HOUR TO SECOND

### C Data Type Definition

```
typedef struct IntrvlHtoS
{
    SMALLINT hour;
    SMALLINT minute;
    DECIMAL4 seconds;
} IntrvlHtoS;
```

### Usage

The *seconds* member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds.

The range of values defined for the SQL INTERVAL HOUR TO SECOND type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL HOUR TO SECOND type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL HOUR TO SECOND in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL HOUR(2) TO SECOND(2)) RETURNS INTERVAL HOUR TO SECOND ...;</pre>	<pre>void f1( IntrvlHtoS *a,         IntrvlHtoS *result,         ... ) { ... }</pre>

## INTERVAL MINUTE

### C Data Type Definition

```
typedef SMALLINT MINUTE;
```

### Usage

The range of values defined for the SQL INTERVAL MINUTE type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL MINUTE type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL MINUTE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL MINUTE ) RETURNS INTERVAL MINUTE ...;</pre>	<pre>void f1( MINUTE  *a,         MINUTE  *result,         ... ) { ... }</pre>

## INTERVAL MINUTE TO SECOND

### C Data Type Definition

```
typedef struct IntrvlMtoS
{
    DECIMAL4 seconds;
    SMALLINT minute;
} IntrvlMtoS;
```

### Usage

The *seconds* member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds.

The range of values defined for the SQL INTERVAL MINUTE TO SECOND type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL MINUTE TO SECOND type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL MINUTE TO SECOND in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL MINUTE TO SECOND ) RETURNS INTERVAL MINUTE TO SECOND ...;</pre>	<pre>void f1( IntrvlMtoS *a,         IntrvlMtoS *result,         ... ) { ... }</pre>

## INTERVAL MONTH

### C Data Type Definition

```
typedef SMALLINT INTERVAL_MONTH;
```

## Usage

The range of values defined for the SQL INTERVAL MONTH type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL MONTH type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL MONTH in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL MONTH ) RETURNS INTERVAL MONTH(4) ...;</pre>	<pre>void f1( INTERVAL_MONTH *a,         INTERVAL_MONTH *result,         ... ) { ... }</pre>

## INTERVAL SECOND

### C Data Type Definition

```
typedef struct IntrvlSec
{
    DECIMAL4 fract_sec;
    SMALLINT whole_sec;
} IntrvlSec;
```

## Usage

The *fract\_sec* member is laid out as a DECIMAL(8,6) numeric field and only contains the fractional seconds. The *whole\_sec* member contains the number of whole seconds.

The range of values defined for the SQL INTERVAL SECOND type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL SECOND type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL SECOND in a UDF definition and IntrvlSec in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL SECOND(4,6) ) RETURNS INTERVAL SECOND ...;</pre>	<pre>void f1( IntrvlSec *a,         IntrvlSec *result,         ... ) { ... }</pre>



## INTERVAL YEAR

### C Data Type Definition

```
typedef SMALLINT INTERVAL_YEAR;
```

### Usage

The range of values defined for the SQL INTERVAL YEAR type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL YEAR type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

This example uses INTERVAL YEAR in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL YEAR ) RETURNS INTERVAL YEAR ...;</pre>	<pre>void f1( INTERVAL_YEAR *a,         INTERVAL_YEAR *result,         ... ) { ... }</pre>

## INTERVAL YEAR TO MONTH

### C Data Type Definition

```
typedef struct IntrvlYtoM
{
  SMALLINT year;
  SMALLINT month;
} IntrvlYtoM;
```

### Usage

The range of values defined for the SQL INTERVAL YEAR TO MONTH type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the INTERVAL YEAR TO MONTH type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using INTERVAL YEAR TO MONTH in a UDF definition and IntrvlYtoM in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A INTERVAL YEAR(4) TO MONTH)</pre>	<pre>void f1( IntrvlYtoM *a,         IntrvlYtoM *result,</pre>

SQL Function Definition	Equivalent C Function Declaration
<pre>RETURNS INTERVAL YEAR TO MONTH ...;</pre>	<pre>... ) { ... }</pre>

## JSON

### C Data Type Definition

```
typedef int JSON_HANDLE;
```

### Usage

Use the JSON\_HANDLE data type when passing a JSON instance as an argument or returning a JSON type result.

Here is an example using a JSON parameter in a UDF definition and JSON\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A JSON(100)) RETURNS JSON(100) ...;</pre>	<pre>void f1( JSON_HANDLE *a,         JSON_HANDLE *result,         ... ) { ... }</pre>

## NUMBER

### C Data Type Definition

```
typedef struct NUMBER
{
  int length;
  short scale;
  BYTE mantissa[NUMBER_MAX_MANTISSA_SIZE];
} NUMBER;
```

### Usage

The layout of NUMBER is the same as that of the NUMBER data type when transferred to or from the client: a two byte scale followed by the significand. The maximum length is 19.

Here is an example using NUMBER in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1(   A NUMBER(30,2) ) RETURNS NUMBER ...;</pre>	<pre>void f1( NUMBER *a,         NUMBER *result,         ... ) { ... }</pre>

# PERIOD (DATE) / PERIOD(TIME) / PERIOD(TIMESTAMP)

## C Data Type Definition

```
typedef int PDT_HANDLE;
```

## Usage

Use the PDT\_HANDLE data type when passing or returning a PERIOD(DATE), PERIOD(TIME(*n*)), PERIOD(TIME(*n*) WITH TIME ZONE), PERIOD(TIMESTAMP(*n*)), or PERIOD(TIMESTAMP(*n*) WITH TIME ZONE)) argument or return type.

The fractional seconds precision defined for the SQL PERIOD(TIME(*n*)), PERIOD(TIME(*n*) WITH TIME ZONE), PERIOD(TIMESTAMP(*n*)), or PERIOD(TIMESTAMP(*n*) WITH TIME ZONE)) type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error.

All PERIOD(TIME(*n*)) and PERIOD(TIMESTAMP(*n*)) values passed to a UDF are in Universal Coordinated Time (UTC), and all PERIOD(TIME(*n*)) and PERIOD(TIMESTAMP(*n*)) values that are returned from a UDF must be in UTC.

The PERIOD(TIME(*n*) WITH TIME ZONE) and PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) types have time zone fields, zone\_hour and zone\_minute, that have an internal form and an external form. When input arguments with these types are passed to a UDF, the values must be in internal form. Similarly, return arguments with these types must be in internal form.

To modify the time zone fields of the PERIOD(TIME(*n*) WITH TIME ZONE) and PERIOD(TIMESTAMP(*n*) WITH TIME ZONE) types, use the following macros in the sqltypes\_td.h file to convert between the internal and external representation of the time zone fields of ANSI\_Time\_Wzone and ANSI\_TimeStamp\_Wzone:

```
TIMEZONE_INTERNAL_TO_EXTERNAL(i, s, h, m)

TIMEZONE_EXTERNAL_TO_INTERNAL(s, h, m, i)
```

*i*  
 Pointer to either an ANSI\_Time\_Wzone or ANSI\_TimeStamp\_Wzone.

**s**

BYTEINT value to store the sign of *i*: 0 if *h* and *m* are negative and 1 if they are positive.

**h**

BYTEINT value to store the hour offset of *i*.

The external form for the hour offset is a BYTEINT value in the range [0, 14], prior and post conversion, which represents the hour offset between UTC and the given time zone.

**m**

BYTEINT value to store the minute offset of *i*.

The external form for the minute offset is a BYTEINT value in the range [0, 59], prior and post conversion, which represents the minute offset between UTC and the given time zone.

The largest value for negative offsets is -12:59. The largest value for positive offsets is +14:00.

This representation results in two representations of UTC, that is +00:00 and -00:00. This mirrors the internal representation which uses both representations for display purposes.

For details on Period types, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using a PERIOD(DATE) in a UDF definition and PDT\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A PERIOD(DATE) )   RETURNS PERIOD(DATE)   ...;</pre>	<pre>void f1( PDT_HANDLE *a,         PDT_HANDLE *result,         ... ) { ... }</pre>

## SMALLINT

### C Data Type Definition

```
typedef short SMALLINT;
```

### Usage

This example uses SMALLINT in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A SMALLINT )</pre>	<pre>void f1( SMALLINT *a,         SMALLINT *result,</pre>

SQL Function Definition	Equivalent C Function Declaration
<pre>RETURNS SMALLINT ...;</pre>	<pre>... ) { ... }</pre>

## TD\_ANYTYPE

### Usage

A TD\_ANYTYPE parameter type can accept any system-defined data type or user-defined type; therefore, the parameters are passed in as void \*.

For details on the TD\_ANYTYPE type, see [Defining Functions that Use the TD\\_ANYTYPE Type](#).

Here is an example using TD\_ANYTYPE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A TD_ANYTYPE ) RETURNS TD_ANYTYPE ...;</pre>	<pre>void f1( void *a,         void *result,         ... ) { ... }</pre>

## TIME

### C Data Type Definition

```
typedef struct ANSI_Time
{
    DECIMAL4 seconds;
    BYTEINT  hour;
    BYTEINT  minute;
} ANSI_Time;
```

### Usage

The *seconds* member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds.

The range of values defined for the SQL TIME type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error.

All TIME values passed to a UDF are in Universal Coordinated Time (UTC), and all TIME values that are returned from a UDF must be in UTC.

For details on the TIME type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example using TIME in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A TIME ) RETURNS TIME ...;</pre>	<pre>void f1( ANSI_Time *a,         ANSI_Time *result,         ... ) { ... }</pre>

## TIME WITH TIME ZONE

### C Data Type Definition

```
typedef struct ANSI_Time_WZone
{
    DECIMAL4 seconds;
    BYTEINT hour;
    BYTEINT minute;
    BYTEINT zone_hour;
    BYTEINT zone_minutes;
} ANSI_Time_WZone;
```

### Usage

The seconds member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds.

The range of values defined for the SQL TIME WITH TIME ZONE type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error.

All TIME WITH TIME ZONE values passed to a UDF are in Universal Coordinated Time (UTC), and all TIME WITH TIME ZONE values that are returned from a UDF must be in UTC.

The TIME WITH TIME ZONE type has time zone fields, zone\_hour and zone\_minute, that have an internal form and an external form. When input arguments of this type are passed to a UDF, the values must be in internal form. Similarly, return arguments with this type must be in internal form.

To modify the time zone fields of the TIME WITH TIME ZONE type, use the following macros in the sqltypes\_td.h file to convert between the internal and external representation of the time zone fields of ANSI\_Time\_Wzone and ANSI\_TimeStamp\_Wzone.

```
TIMEZONE_INTERNAL_TO_EXTERNAL(i, s, h, m)
```

```
TIMEZONE_EXTERNAL_TO_INTERNAL(s, h, m, i)
```

*i*

Pointer to either an ANSI\_Time\_Wzone or ANSI\_TimeStamp\_Wzone.

*s*

BYTEINT value to store the sign of *i*: 0 if *h* and *m* are negative and 1 if the they are positive.

*h*

BYTEINT value to store the hour offset of *i*.

The external form for the hour offset is a BYTEINT value in the range [0, 14], prior and post conversion, which represents the hour offset between UTC and the given time zone.

*m*

BYTEINT value to store the minute offset of *i*.

The external form for the minute offset is a BYTEINT value in the range [0, 59], prior and post conversion, which represents the minute offset between UTC and the given time zone.

The largest value for negative offsets is -12:59. The largest value for positive offsets is +14:00.

This representation results in two representations of UTC, that is +00:00 and -00:00. This mirrors the internal representation which uses both representations for display purposes.

For details on the TIME WITH TIME ZONE type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Here is an example of TIME WITH TIME ZONE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A TIME WITH TIME ZONE)   RETURNS TIME WITH TIME ZONE   ...;</pre>	<pre>void f1( ANSI_Time_WZone *a,         ANSI_Time_WZone *result,         ... ) { ... }</pre>

## TIMESTAMP

### C Data Type Definition

```
typedef struct TimeStamp  
{  
    DECIMAL4 seconds;  
    SMALLINT year;  
    BYTEINT month;  
    BYTEINT day;
```

```

    BYTEINT  hour;
    BYTEINT  minute;
} TimeStamp;

```

## Usage

The *seconds* member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds.

The range of values defined for the SQL TIMESTAMP type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the TIMESTAMP type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

All TIMESTAMP values passed to a UDF are in Universal Coordinated Time (UTC), and all TIMESTAMP values that are returned from a UDF must be in UTC.

Here is an example using TIMESTAMP in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre> CREATE FUNCTION F1 (   A TIMESTAMP ) RETURNS TIMESTAMP ...; </pre>	<pre> void f1( TimeStamp *a,         TimeStamp *result,         ... ) { ... } </pre>

## TIMESTAMP WITH TIME ZONE

### C Data Type Definition

```

typedef struct ANSI_TimeStamp_WZone
{
    DECIMAL4 seconds;
    SMALLINT year;
    BYTEINT month;
    BYTEINT day;
    BYTEINT hour;
    BYTEINT minute;
    BYTEINT zone_hour;
    BYTEINT zone_minutes;
} ANSI_TimeStamp_WZone;

```

## Usage

The *seconds* member is laid out as a DECIMAL(8,6) numeric field, which can represent up to two digits of whole seconds and six digits of fractional seconds.



The range of values defined for the SQL `TIMESTAMP WITH TIME ZONE` type applies to the input arguments and return argument of a function. A value outside the valid range of values produces an error. For details on the `TIMESTAMP WITH TIME ZONE` type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

All `TIMESTAMP WITH TIME ZONE` values passed to a UDF are in Universal Coordinated Time (UTC), and all `TIMESTAMP WITH TIME ZONE` values that are returned from a UDF must be in UTC.

The `TIMESTAMP WITH TIME ZONE` type has time zone fields, `zone_hour` and `zone_minute`, that have an internal form and an external form. When input arguments of this type are passed to a UDF, the values must be in internal form. Similarly, return arguments with this type must be in internal form.

To modify the time zone fields of the `TIMESTAMP WITH TIME ZONE` type, use the following macros in the `sqltypes_td.h` file to convert between the internal and external representation of the time zone fields of `ANSI_Time_Wzone` and `ANSI_TimeStamp_Wzone`.

```
TIMEZONE_INTERNAL_TO_EXTERNAL(i, s, h, m)
```

```
TIMEZONE_EXTERNAL_TO_INTERNAL(s, h, m, i)
```

***i***

Pointer to either an `ANSI_Time_Wzone` or `ANSI_TimeStamp_Wzone`.

***s***

BYTEINT value to store the sign of *i*: 0 if *h* and *m* are negative and 1 if they are positive.

***h***

BYTEINT value to store the hour offset of *i*.

The external form for the hour offset is a BYTEINT value in the range [0, 14], prior and post conversion, which represents the hour offset between UTC and the given time zone.

***m***

BYTEINT value to store the minute offset of *i*.

The external form for the minute offset is a BYTEINT value in the range [0, 59], prior and post conversion, which represents the minute offset between UTC and the given time zone.

The largest value for negative offsets is -12:59. The largest value for positive offsets is +14:00.

This representation results in two representations of UTC, that is +00:00 and -00:00. This mirrors the internal representation which uses both representations for display purposes.

Here is an example using `TIMESTAMP WITH TIME ZONE` in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A TIMESTAMP WITH TIME ZONE) RETURNS INTEGER ...;</pre>	<pre>void f1( ANSI_TimeStamp_WZone *a,         INTEGER *result,         ... ) { ... }</pre>

## ***UDT\_name* / VARIANT\_TYPE**

### **C Data Type Definition**

```
typedef int UDT_HANDLE;
```

### **Usage**

Use the UDT\_HANDLE data type when passing a distinct or structured (including dynamic) UDT argument or returning a distinct or structured UDT.

Here is an example using a UDT in a UDF definition and UDT\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A circleUDT ) RETURNS circleUDT ...;</pre>	<pre>void f1( UDT_HANDLE *a,         UDT_HANDLE *result,         ... ) { ... }</pre>

Here is an example using a dynamic UDT in a UDF definition and UDT\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A VARIANT_TYPE ) RETURNS INTEGER ...;</pre>	<pre>void f1( UDT_HANDLE *a,         INTEGER *result,         ... ) { ... }</pre>

## **VARBYTE**

### **C Data Type and Macro Definition**

```
typedef struct VARBYTE
{
  int    length;          /* length of string */
};
```

```

    BYTE  bytes[1];          /* bytes - size must be adjusted */
} VARBYTE;
#define VARBYTE_M(len) struct { int length; BYTE bytes[len]; }

```

## Usage

Use the VARBYTE structure for reference to an existing VARBYTE.

Use the VARBYTE\_M macro to define your own VARBYTE of specific length. For example, to define a VARBYTE of length 30, use the macro like this:

```
VARBYTE_M(30) myvbstr;
```

If the function result type is variable length byte data, then the result argument points to a data area containing a VARBYTE structure with a bytes field reserved to the maximum defined by the RETURNS clause in the CREATE FUNCTION statement.

Here is an example using VARBYTE in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre> CREATE FUNCTION F1 (   A VARBYTE(30) ) RETURNS VARBYTE(12) ...; </pre>	<pre> void f1( VARBYTE *a,         VARBYTE *result,         ... ) { ... } </pre>

## VARCHAR CHARACTER SET GRAPHIC, LONG VARCHAR CHARACTER SET GRAPHIC

### C Data Type and Macro Definition

```

typedef struct VARGRAPHIC {
    int      length;          /* length of string */
    GRAPHIC  graphic[1];     /* string - size must be adjusted */
} VARGRAPHIC;

#define VARGRAPHIC_M(len) struct { int length; GRAPHIC graphic[len]; }

```

## Usage

Use the VARGRAPHIC structure for reference to an existing VARCHAR CHARACTER SET GRAPHIC value.

Use the VARGRAPHIC\_M macro to define your own VARCHAR CHARACTER SET GRAPHIC value of specific length. For example, to define a VARCHAR CHARACTER SET GRAPHIC value of length 30, use the macro like this:

```
VARGRAPHIC_M(30) myvgstr;
```

If the function result type is variable length graphic data, then the result argument points to a data area containing a VARGRAPHIC structure with a graphic field reserved to the maximum defined by the RETURNS clause in the CREATE FUNCTION statement.

Here is an example using VARCHAR(n) CHARACTER SET GRAPHIC in a UDF definition and C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (   A VARCHAR(30) CHARACTER SET   GRAPHIC )   RETURNS VARCHAR(12) CHARACTER   SET GRAPHIC   ...;</pre>	<pre>void f1( VARGRAPHIC *a,         VARGRAPHIC *result,         ... ) { ... }</pre>

## XML

### C Data Type Definition

```
typedef int XML_HANDLE;
```

### Usage

Use the XML\_HANDLE data type when passing an XML instance as an argument or returning an XML type result.

Here is an example using an XML parameter in a UDF definition and XML\_HANDLE in a C function declaration.

SQL Function Definition	Equivalent C Function Declaration
<pre>CREATE FUNCTION F1 (A XML(100))   RETURNS XML(100)   ...;</pre>	<pre>void f1( XML_HANDLE *a,         XML_HANDLE *result,         ... ) { ... }</pre>

UDFs, UDMs, and external stored procedures can also process XML data as VARCHAR, CLOB, VARBYTE, or BLOB values. You can use the XMLSERIALIZE function to serialize an XML value to a VARCHAR, CLOB, VARBYTE, or BLOB value before passing it to an external routine. You can use the CREATEXML function to create an XML type value from the results of an external routine.

For information about the XMLSERIALIZE and CREATEXML functions, see *Teradata Vantage™ - XML Data Type*, B035-1140.

Therefore, in addition to the XML\_HANDLE data type, you can also use the following C data types as parameters and return types of UDFs, UDMs, and external stored procedures that process XML:

- [CHARACTER VARYING / LONG VARCHAR / VARCHAR](#)
- [CHARACTER LARGE OBJECT / CLOB](#)
- [VARBYTE](#)
- [BINARY LARGE OBJECT / BLOB](#)

## Java Data Types

### Supported Types

This section identifies the SQL data types that are supported for parameters of Java UDFs and external stored procedures and how the SQL data types map to Java data types.

The default parameter mapping convention is simple mapping, where SQL data types map to Java primitives (identified by *Simple Map*). When an appropriate primitive does not exist, the SQL data types map to Java classes (identified by *Object Map*).

For UDFs or external stored procedures that allow parameters to pass in or return nulls, simple mapping to Java primitives is not appropriate.

For some types, more than one type of mapping is available. The default mapping is identified along with the additional mappings. The Java external routine writer can choose the Java data type to which the SQL type will be mapped depending on the requirement.

To override the default mapping, the EXTERNAL NAME clause in the CREATE/REPLACE FUNCTION or CREATE/REPLACE PROCEDURE statement must explicitly specify the mapping in the parameter list of the Java method.

### Example: Distinct UDT with Default Mapping

In this example, the Java routine declaration in the EXTERNAL NAME clause does not include the parameter list. Therefore, simple mapping (the default) is used for the MyInt UDT.

```
CREATE TYPE MYINT AS INTEGER FINAL;

REPLACE FUNCTION MyScore(A1 MYINT)
RETURNS MYINT
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.myScore';
```

Equivalent Java Method:

```
public static int myScore(int a) throws SQLException
```

### Example: ST\_Geometry with Nondefault Mapping

This example shows a Java UDF with an ST\_Geometry parameter using a nondefault mapping to an ST\_Geometry class.

```
REPLACE FUNCTION get_Geom(G1 ST_GEOMETRY)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME
'UDF_JAR:UserDefinedFunctions.get_geom(com.teradata.fnc.ST_Geometry)
returns int';
```

```
public static int get_geom(com.teradata.fnc.ST_Geometry g1)
    throws SQLException
```

### ARRAY/VARRAY

The ARRAY/VARRAY SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.sql.Array

The com.teradata.fnc.Array class implements the java.sql.Array interface.

Both 1-D and N-D arrays are supported as parameters and return types for Java external stored procedures and UDFs.

### BIGINT

The BIGINT SQL data type maps to the following Java data types.

Simple Map	Object Map
long (J)	java.lang.Long

Example SQL function definition (default mapping, no null values):

```
CREATE PROCEDURE F1 (IN A BIGINT, OUT B BIGINT)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(long a, long[] b)
{ ... }
```

Example SQL function definition (allows null values):

```
CREATE PROCEDURE F1 (IN A BIGINT, OUT B BIGINT)
LANGUAGE JAVA
...
EXTERNAL NAME 'j1:c1.f1(java.lang.Long, java.lang.Long[])';
```

Equivalent Java method:

```
public static void f1(Long a, Long[] b)
{ ... }
```

## BLOB (Binary Large Object)

The BLOB SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.sql.Blob

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A BLOB, OUT B BLOB)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Blob a, java.sql.Blob[] b)
{ ... }
```

## BYTE

The BYTE SQL data type maps to the following Java data type.

Simple Map	Object Map
None	byte[] ([B)

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A BYTE[30], OUT B BYTE[30])
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(byte[] a, byte[][] b)
{ ... }
```

## BYTEINT

The BYTEINT SQL data type maps to the following Java data types.

Simple Map	Object Map
byte (B)	java.lang.Byte

Example SQL function definition:

```
CREATE PROCEDURE F1 (IN A BYTEINT, OUT B BYTEINT)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(byte a, byte[] b)
{ ... }
```

Example SQL function definition (allows null values):

```
CREATE PROCEDURE F1 (IN A BYTEINT, OUT B BYTEINT)
LANGUAGE JAVA
...
EXTERNAL NAME 'j1:c1.f1(java.lang.Byte, java.lang.Byte[])';
```

Equivalent Java method:

```
public static void f1(Byte a, Byte[] b)
{ ... }
```

## CHARACTER/CHAR

The CHARACTER SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.lang.String

Example SQL function definition (default mapping, allows null values):



```
CREATE PROCEDURE F1 (IN A CHAR(30), OUT B CHAR(30))
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(String a, String[] b)
{ ... }
```

## CLOB (Character Large Object)

The CLOB SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.sql.Clob

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A CLOB, OUT B CLOB)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Clob a, java.sql.Clob[] b)
{ ... }
```

## DATASET

The DATASET SQL data type maps to the following Java data types.

Storage Format	Simple Map	Object Map
DATASET STORAGE FORMAT AVRO	None	<ul style="list-style-type: none"> <li>java.sql.Blob (default mapping)</li> <li>byte[]</li> </ul>
DATASET STORAGE FORMAT CSV	None	<ul style="list-style-type: none"> <li>java.sql.Clob (default mapping)</li> <li>java.lang.String</li> </ul>

For DATASET types in the Avro storage format, the default mapping is to java.sql.Blob. An additional mapping is provided to map the DATASET type to byte[]. You can use this mapping when the DATASET size will not exceed 64K. This can provide better performance than LOB-based parameter mappings.

The DATASET type in the Avro storage format is converted into its transform format, which is the schema defined for the instance, encoded in UTF-8 and null-terminated, followed by the binary-encoded Avro value.

For DATASET types in the CSV storage format, the default mapping is to `java.sql.Clob`. An additional mapping is provided to map the DATASET type to `java.lang.String`. You can use this mapping when the DATASET size will not exceed 64K. This can provide better performance than LOB-based parameter mappings.

Note that when DATASET types in the CSV storage format are mapped to `java.sql.Clob` or `java.lang.String`, only the CSV data is included, and not any optional schema.

Example SQL function definition:

```
CREATE PROCEDURE F1 (IN A DATASET(8000) Storage Format Avro,
                    OUT B DATASET(8000) Storage Format Avro)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Blob a, java.sql.Blob[] b)
{ ... }
```

## DATE

The DATE SQL data type maps to the following Java data type.

Simple Map	Object Map
None	<code>java.sql.Date</code>

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A DATE, OUT B DATE)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Date a, java.sql.Date[] b)
{ ... }
```

## DECIMAL/NUMERIC

The DECIMAL/NUMERIC SQL data type maps to the following Java data type.

Simple Map	Object Map
None	<code>java.math.BigDecimal</code>

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A DECIMAL(8,2), OUT B DECIMAL(8,2))
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(BigDecimal a, BigDecimal[] b)
{ ... }
```

## FLOAT / DOUBLE PRECISION / REAL

The FLOAT / DOUBLE PRECISION / REAL SQL data type maps to the following Java data types.

Simple Map	Object Map
double (D)	java.lang.Double

Example SQL function definition:

```
CREATE PROCEDURE F1 (IN A FLOAT, OUT B FLOAT)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(double a, double[] b)
{ ... }
```

Example SQL function definition (allows null values):

```
CREATE PROCEDURE F1 (IN A FLOAT, OUT B FLOAT)
LANGUAGE JAVA
...
EXTERNAL NAME 'j1:c1.f1(java.lang.Double, java.lang.Double[])';
```

Equivalent Java method:

```
public static void f1(Double a, Double[] b)
{ ... }
```

## Geospatial Data Type: MBB

The Geospatial MBB SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>• java.lang.String (default mapping)</li> <li>• double[]</li> </ul>

The MBB type maps to `java.lang.String` by default. The format is (Xmin,Ymin,Zmin,Xmax,Ymax,Zmax).

An additional mapping is provided to map MBB to a primitive double array, `double[]`. The array would contain 6 double values that correspond to the Xmin, Ymin, Zmin, Xmax, Ymax, and Zmax values.

When the MBB type is a structured UDT attribute, the default mapping (`java.lang.String`) will apply. The MBB type can be at any nested level.

### Geospatial Data Type: MBR

The Geospatial MBR SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>• <code>java.lang.String</code> (default mapping)</li> <li>• <code>double[]</code></li> </ul>

The MBR type is mapped to `java.lang.String` by default. The format is (Xmin,Ymin,Xmax,Ymax).

An additional mapping is provided to map MBR to a primitive double array, `double[]`. The array would contain 4 double values that correspond to the Xmin, Ymin, Xmax, and Ymax values.

When the MBR type is a structured UDT attribute, the default mapping (`java.lang.String`) will apply. The MBR type can be at any nested level.

### Geospatial Data Type: ST\_Geometry

The Geospatial ST\_Geometry SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>• <code>java.sql.Clob</code> (default mapping)</li> <li>• <code>java.sql.Blob</code></li> <li>• <code>com.teradata.fnc.ST_Geometry</code></li> <li>• <code>java.lang.String</code></li> <li>• <code>byte[]</code></li> </ul>

The ST\_Geometry type is converted to its Well-Known Text (WKT) representation and passed as a Clob to the Java routine. The Clob value passed to the Java routine will not contain the spatial reference identifier (SRID) that is contained within a geometry object.

A mapping is provided to map the ST\_Geometry type to `java.sql.Blob`. The Geospatial type is converted to its Well-Known Binary (WKB) representation and passed as a Blob to the Java routine. The Blob will not contain the SRID that is contained within the geometry object.

A mapping is provided to map the ST\_Geometry type to the `com.teradata.fnc.ST_Geometry` class. The ST\_Geometry object can be used to retrieve the WKT, WKB, and SRID of the geometry value or set the geometry value to a WKT or WKB value (with supplied SRID). Note that this is the only mapping that can read or set the SRID value.

Non-LOB-based mappings are provided to `java.lang.String` (WKT representation in LATIN character set) and `byte[]` (WKB representation). These mappings can be used when it is known that the `ST_Geometry` parameter size will not exceed 64K. This may provide better performance than LOB-based parameter mappings.

Structured UDTs with `ST_Geometry` attributes cannot be passed to Java UDFs or external stored procedures.

The performance benefits in sending the geometry as a BLOB rather than a CLOB are as follows:

- The geometry is stored internally in the WKB format in the field so there is no requirement to convert it to the WKT format for a CLOB.
- The user would have to parse the text form in order to process the geometry data. The binary format can be easily traversed without the need to parse text.
- The binary form contains floats that represent coordinate values versus strings that represent them in the text format; therefore, the coordinate values may have a little better precision in the binary format.
- The WKB format is smaller than the WKT format.

This example shows a Java UDF with an `ST_Geometry` parameter using a nondefault mapping to an `ST_Geometry` class.

```
REPLACE FUNCTION get_Geom(G1 ST_GEOMETRY)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME
'UDF_JAR:UserDefinedFunctions.get_geom(com.teradata.fnc.ST_Geometry)
returns int';
```

```
public static int get_geom(com.teradata.fnc.ST_Geometry g1)
    throws SQLException
```

## INTEGER

The `INTEGER` SQL data type maps to the following Java data types.

Simple Map	Object Map
<code>int (I)</code>	<code>java.lang.Integer</code>

Example SQL function definition:

```
CREATE PROCEDURE F1 (IN A INTEGER, OUT B INTEGER)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(int a, int[] b)
{ ... }
```

Example SQL function definition (allows null values):

```
CREATE PROCEDURE F1 (IN A INTEGER, OUT B INTEGER)
LANGUAGE JAVA
...
EXTERNAL NAME 'j1:c1.f1(java.lang.Integer, java.lang.Integer[])';
```

Equivalent Java method:

```
public static void f1(Integer a, Integer[] b)
{ ... }
```

**Interval SQL Data Types**

The Interval SQL data types listed below map to the following Java data type:

- INTERVAL DAY
- INTERVAL DAY TO HOUR
- INTERVAL DAY TO MINUTE
- INTERVAL DAY TO SECOND
- INTERVAL HOUR
- INTERVAL HOUR TO MINUTE
- INTERVAL HOUR TO SECOND
- INTERVAL MINUTE
- INTERVAL MINUTE TO SECOND
- INTERVAL MONTH
- INTERVAL SECOND
- INTERVAL YEAR
- INTERVAL YEAR TO MONTH

Simple Map	Object Map
None	java.lang.String

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A INTERVAL MONTH, OUT B INTERVAL MONTH)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(String a, String[] b)
{ ... }
```

## JSON

The JSON SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>• java.sql.Clob (default mapping)</li> <li>• java.sql.Blob</li> <li>• java.lang.String</li> <li>• byte[]</li> </ul>

The default mapping is to java.sql.Clob.

A mapping to java.sql.Blob is provided that will send the data in binary JSON format.

Non-LOB-based mappings are provided to java.lang.String (text representation) and byte[] (BSON format). These mappings can be used when it is known that the JSON parameter size will not exceed 64K. This may provide better performance than LOB-based parameter mappings.

Structured UDTs with JSON attributes are allowed as parameters to Java UDFs at any nested level. The JSON value will be mapped as an inline value to a String or a byte[]. Lob mappings are not supported.

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A JSON(20) CHARACTER SET LATIN, OUT B JSON(40)
CHARACTER SET LATIN)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Clob a, java.sql.Clob[] b)
{ ... }
```

## NUMBER

The NUMBER SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.math.BigDecimal

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (INOUT A NUMBER(30,2))
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(BigDecimal[] a)
{ ... }
```

## Period

Period SQL data types map to the following Java data type.

Simple Map	Object Map
None	java.sql.Struct

A Period data type is mapped to java.sql.Struct. The com.teradata.fnc.Struct class implements the java.sql.Struct interface.

The Period Struct objects are constructed as follows:

Period (Date)

```
java.sql.Struct{
    java.sql.Date,
    java.sql.Date
}
```

Period(Time)

```
java.sql.Struct{
    java.sql.Time,
    java.sql.Time
}
```

Period(Time With Time Zone)

```
java.sql.Struct{
    java.sql.Struct{
        java.sql.Time,
        java.util.Calendar,
    }
    java.sql.Struct{
        java.sql.Time,
        java.util.Calendar
    }
}
```



```
    }  
}
```

Period(Timestamp)

```
java.sql.Struct{  
    java.sql.Timestamp,  
    java.sql.Timestamp  
}
```

Period(Timestamp With Time Zone)

```
java.sql.Struct{  
    java.sql.Struct{  
        java.sql.Timestamp,  
        java.util.Calendar,  
    }  
    java.sql.Struct{  
        java.sql.Timestamp,  
        java.util.Calendar  
    }  
}
```

For Period(Time With Time Zone) and Period(Timestamp With Time Zone), the Time or Timestamp object in the Struct stores the local time of the input Time/Timestamp value. The Calendar object stores only the time zone part of the input Time/Timestamp value. Other information such as hour, minute, second, and so forth is not set in the Calendar object based on the input Time/Timestamp value. Instead this information is set to the default local time when the Calendar object is created.

Period types can be attributes of a structured UDT at any level.

**SMALLINT**

The SMALLINT SQL data type maps to the following Java data types.

Simple Map	Object Map
short (S)	java.lang.Short

Example SQL function definition:

```
CREATE PROCEDURE F1 (IN A SMALLINT, OUT B SMALLINT)  
LANGUAGE JAVA  
...;
```

Equivalent Java method:

```
public static void f1(short a, short[] b)
{ ... }
```

Example SQL function definition (allows null values):

```
CREATE PROCEDURE F1 (IN A SMALLINT, OUT B SMALLINT)
LANGUAGE JAVA
...
EXTERNAL NAME 'j1:c1.f1(java.lang.Short, java.lang.Short[])';
```

Equivalent Java method:

```
public static void f1(Short a, Short[] b)
{ ... }
```

## TD\_ANYTYPE

The TD\_ANYTYPE SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.lang.Object

User-defined types (UDTs) and complex data types (CDTs) can be passed to a Java UDF or external stored procedure with TD\_ANYTYPE parameters. These will be mapped to java.lang.Object in the Java routine. The object type will always correspond to the default mapping type. For instance, if a JSON value is passed to a Java UDF with a TD\_ANYTYPE parameter, the Java routine will receive a java.sql.Clob object.

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A TD_ANYTYPE, INOUT B TD_ANYTYPE)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.lang.Object a, java.lang.Object[] b)
{ ... }
```

## TIME

The TIME SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.sql.Time

Simple Map	Object Map
	<b>Note:</b> Does not permit the full granularity of nanosecond time provided by the SQL type

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A TIME, OUT B TIME)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Time a, java.sql.Time[] b)
{ ... }
```

## TIME WITH TIME ZONE

The TIME WITH TIME ZONE SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>• java.sql.Time (default mapping)</li> <li>• java.sql.Struct</li> </ul>

The TIME WITH TIME ZONE type maps to java.sql.Time by default. The following apply:

- This mapping does not permit the full granularity of nanosecond time provided by the SQL type.
- This mapping does not preserve the time zone offset.

A mapping is provided which maps TIME WITH TIME ZONE to java.sql.Struct. Mapping to java.sql.Struct allows the time zone value to be passed to the Java routine. The com.teradata.fnc.Struct class implements the java.sql.Struct interface.

The Time with Time Zone Struct object is constructed as follows:

```
Time with Time Zone
  java.sql.Struct{
    java.sql.Time,
    java.util.Calendar
  }
```

The Time object in the Struct stores the local time of the input Time value. The Calendar object stores only the time zone part of the input Time value. Other information such as hour, minute, second, and so forth is not set in the Calendar object based on the input Time value. Instead this information is set to the default local time when the Calendar object is created.

If the TIME WITH TIME ZONE type is a structured UDT attribute or an Array element type, then the default mapping (java.sql.Time) will apply.

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A TIME WITH TIME ZONE, OUT B TIME WITH TIME ZONE)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Struct a, java.sql.Struct[] b)
{ ... }
```

## TIMESTAMP

The TIMESTAMP SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.sql.Timestamp

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A TIMESTAMP, OUT B TIMESTAMP)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Timestamp a, java.sql.Timestamp[] b)
{ ... }
```

## TIMESTAMP WITH TIME ZONE

The TIMESTAMP WITH TIME ZONE SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>java.sql.Timestamp (default mapping)</li> <li>java.sql.Struct</li> </ul>

The TIMESTAMP WITH TIME ZONE type maps to java.sql.Timestamp by default. This mapping does not preserve the time zone offset.

A mapping is provided which maps TIMESTAMP WITH TIME ZONE to java.sql.Struct. Mapping to java.sql.Struct allows the time zone value to be passed to the Java routine. The com.teradata.fnc.Struct class implements the java.sql.Struct interface.

The Timestamp with Time Zone Struct object is constructed as follows:

```
Timestamp with Time Zone
  java.sql.Struct{
    java.sql.Timestamp,
    java.util.Calendar
  }
```

The Timestamp object in the Struct stores the local time of the input Timestamp value. The Calendar object stores only the time zone part of the input Timestamp value. Other information such as hour, minute, second, and so forth is not set in the Calendar object based on the input Timestamp value. Instead this information is set to the default local time when the Calendar object is created.

For example, if the input Timestamp value is "2001-02-05 05:13:11.207000 -00:30", then it is mapped to the following Struct:

```
Struct{
  java.sql.Timestamp;
  java.util.Calendar;
}
```

The Timestamp object will be "2001-02-05 05:13:11.207000" of the local time based on the JVM default time zone. The Calendar object will be "GMT-00:30".

If the TIMESTAMP WITH TIME ZONE type is a structured UDT attribute or an Array element type, then the default mapping (java.sql.Timestamp) will apply.

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A TIMESTAMP WITH TIME ZONE, OUT B TIMESTAMP WITH
TIME ZONE)
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(java.sql.Struct a, java.sql.Struct[] b)
{ ... }
```

## UDT (Distinct)

The distinct UDT SQL data type maps to the following Java data types.

Simple Map	Object Map
Primitive Java data type, such as int or short	Java class corresponding to the base data type. For example: <ul style="list-style-type: none"> <li>java.lang.Integer</li> <li>java.lang.Double</li> <li>...</li> </ul>

A distinct UDT is mapped to the Java type (simple or object mapping) corresponding to its predefined type. The distinct UDT is converted to its predefined type before being passed to the Java routine.

The default mapping for a distinct UDT is the simple mapping to the primitive Java data type. That is, if the Java parameter declaration is not specified in the EXTERNAL NAME clause in the CREATE FUNCTION or CREATE PROCEDURE statement, then the UDT distinct types are mapped to the primitive Java data types. Otherwise the mapping is done based on the parameter type in the declaration.

When a distinct UDT is a structured UDT attribute or an Array element type, it will be object mapped to the corresponding Java data type.

In this example, the Java routine declaration in the EXTERNAL NAME clause does not include the parameter list. Therefore, simple mapping (the default) is used for the MyInt UDT.

```
CREATE TYPE MYINT AS INTEGER FINAL;

REPLACE FUNCTION MyScore(A1 MYINT)
RETURNS MYINT
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.myScore';
```

Equivalent Java Method:

```
public static int myScore(int a) throws SQLException
```

**UDT (Structured)**

The structured UDT SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.sql.Struct

A structured UDT is mapped to java.sql.Struct. The com.teradata.fnc.Struct class implements the java.sql.Struct interface.

Structured UDTs with LOBs at a nested level are not supported. LOBs are only allowed at the first level of nesting.

Structured UDTs with XML or ST\_Geometry attributes are not supported.

**VARBYTE**

The VARBYTE SQL data type maps to the following Java data type.

Simple Map	Object Map
None	byte[] ([B)

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A VARBYTE(30), OUT B VARBYTE(30))
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(byte[] a, byte[][] b)
{ ... }
```

## **VARCHAR / CHARACTER VARYING / CHAR VARYING / LONG VARCHAR**

The VARCHAR SQL data type maps to the following Java data type.

Simple Map	Object Map
None	java.lang.String

Example SQL function definition (default mapping, allows null values):

```
CREATE PROCEDURE F1 (IN A VARCHAR(30), OUT B VARCHAR(30))
LANGUAGE JAVA
...;
```

Equivalent Java method:

```
public static void f1(String a, String[] b)
{ ... }
```

## **XML**

The XML SQL data type maps to the following Java data types.

Simple Map	Object Map
None	<ul style="list-style-type: none"> <li>• java.sql.SQLXML (default mapping)</li> <li>• java.lang.String</li> <li>• java.sql.Blob</li> <li>• byte[]</li> </ul>

When mapped to java.sql.SQLXML, the XML type is handled as a CLOB-based type by the Java environment. The com.teradata.fnc.SQLXML class implements the java.sql.SQLXML interface.

A mapping to `java.lang.String` is provided that will send the XML string in UNICODE. The mapping can be used when the size of the XML value is less than 64K.

Mappings are provided to `java.sql.Blob` and `byte[]` which send the XML data in UTF-8 format.

Structured UDTs with XML attributes cannot be passed to Java UDFs or external stored procedures.

## Unsupported Types

The following SQL data types are not supported as parameters or return types of Java UDFs or external stored procedures:

- CHARACTER CHARACTER SET GRAPHIC
- LONG VARCHAR CHARACTER SET GRAPHIC
- VARCHAR CHARACTER SET GRAPHIC
- VARIANT\_TYPE



# C Library Functions

The following sections discuss C library functions that Teradata provides for use by C/C++ UDFs, UDMs, and external stored procedures.

For information on standard C library functions, see [Using Standard C Library Functions](#).

## Function Types

### Aggregate Intermediate Storage

The aggregate library functions allocate the intermediate storage you need for accumulating summary information in an aggregate function.

IF you want to ...	THEN use this library function ...
allocate intermediate storage for accumulating summary information in an aggregate function	FNC_DefMem

### ARRAY Data Type Interface

ARRAY interface functions enable a UDF, UDM, or external stored procedure to access and set the values of the elements in an ARRAY data type.

IF you want to ...	THEN use this library function ...
get the total number of elements in an ARRAY input parameter that are set to a value, including NULL	FNC_GetArrayElementCount
get information such as the element data type, number of dimensions, total number of elements, and scope information about an ARRAY input parameter	FNC_GetArrayTypeInfo_EON
get the number of dimensions defined for an ARRAY input parameter when this information is not known to the external routine	FNC_GetArrayNumDimensions
set either one bit or all bits in a NullBitVector	FNC_SetNullBitVector
check the value of one bit in a NullBitVector	FNC_CheckNullBitVector
set one bit in a NullBitVector where the bit to be set may be referenced by the ARRAY type element index as specified by dimension	FNC_SetNullBitVectorByElemIndex
check the value of one bit in a NullBitVector where the bit to be checked may be referenced by the ARRAY type element index as specified by dimension	FNC_CheckNullBitVectorByElemIndex

IF you want to ...	THEN use this library function ...
get a range of one or more elements of an ARRAY type and for any elements that are either NULL or not present, set the bit to 0 in the NullBitVector	FNC_GetArrayElements
set one or more elements of an ARRAY return parameter to the same value	FNC_SetArrayElements
set the value of one or more elements of an ARRAY return parameter where each element can be set to a different value	FNC_SetArrayElementsWithMultiValues
get one or more UDT_HANDLES that can be used to operate on the elements of an ARRAY data type whose element type is a UDT	FNC_GetUDTHandles

## DATASET Data Type Interface

The following functions enable a UDF or external stored procedure to access and set the value of a DATASET parameter, or to get information about the DATASET type parameter.

IF you want to ...	THEN use this library function ...
get information about a DATASET type instance, such as the maximum length, inline length, and storage format	FNC_GetDatasetInfo
get the schema of a DATASET type instance	FNC_GetDatasetSchema
read the schema of a DATASET type instance, which is stored as a LOB, using the LOB FNC routines	FNC_GetDatasetSchemaLob
get the value of a DATASET type instance when the DATASET data is <i>not</i> stored as a LOB	FNC_GetInternalValue
set the value of a DATASET type instance when the DATASET data will <i>not</i> be stored as a LOB	FNC_SetInternalValue
read DATASET data, which is stored as a LOB, using the LOB FNC routines	FNC_GetDatasetInputLob
write DATASET data to a LOB associated with a DATASET instance	FNC_GetDatasetResultLob
pass a LOB_LOCATOR to a DATASET type instance allowing the instance to use the data referenced by the locator to set its schema and binary-encoded Avro value	FNC_SetDatasetLob

## Geospatial Data Type Interface

The following functions allow you to access, manipulate, and return geospatial data of type ST\_Geometry, MBR, and MBB using external routines.

IF you want to ...	THEN use this library function ...
get the maximum length of an ST_Geometry type and an indication of whether or not it is stored as a LOB	FNC_GetGeometryInfo
get the Well-Known Text (WKT) representation of an inline ST_Geometry value (that is, one that is not stored as a LOB)	FNC_GeomGetWKT
get the WKT representation of an ST_Geometry value that is stored as a LOB	FNC_GeomGetWKTCLob
get the length in bytes of an ST_Geometry WKT	FNC_GeomGetWKTSize
get the LOB locator to a CLOB for an ST_Geometry that is the return value of a UDF/UDM or an INOUT or OUT parameter for an external stored routine	FNC_GeomGetResultWKTCLob
set the value of an ST_Geometry type using a WKT string	FNC_GeomSetWKT
set the value of an ST_Geometry type using a WKT CLOB	FNC_GeomSetWKTCLob
get the Well-Known Binary (WKB) representation of an inline ST_Geometry value (that is, one that is not stored as a LOB)	FNC_GeomGetWKB
get the WKB representation of an ST_Geometry value that is stored as a LOB	FNC_GeomGetWKBBLob
get the length in bytes of an ST_Geometry WKB	FNC_GeomGetWKBSize
get the LOB locator to a BLOB for an ST_Geometry that is the return value of a UDF/UDM or an INOUT or OUT parameter for an external stored procedure	FNC_GeomGetResultWKBBLob
set the value of an ST_Geometry type using a WKB representation of the geometry	FNC_GeomSetWKB
set the value of an ST_Geometry type using a WKB BLOB representation of the geometry	FNC_GeomSetWKBBLob
return a buffer containing four coordinate values that define an MBR (minimum bounding rectangle) type	FNC_MBRGetValue
set the four coordinate values that define an MBR type using a buffer	FNC_MBRSetValue
return a buffer containing six coordinate values that define an MBB (minimum bounding box) type	FNC_MBBGetValue
set the six coordinate values that define an MBB using a buffer	FNC_MBBSetValue

## Global Information

The global information function returns session information related to the current execution of a UDF, UDM, or external stored procedure.

IF you want to ...	THEN use this library function ...
get global information, such as user account, user name, user ID, and session number related to the currently running UDF, UDM, or external stored procedure	FNC_DbsInfo_EON

## GLOP Access

The GLOP access functions provide a way for external routines to map a GLOP set and access the data. Note that you must call FNC\_Get\_GLOP\_Map first before calling any other GLOP function.

IF you want to ...	THEN use this library function ...
get the GLOP set map information associated with an external routine	FNC_Get_GLOP_Map
map different pages of a read-only GLOP or reload the single page of a read/write or globally modifiable GLOP	FNC_GLOP_Map_Page
lock a vproc-local GLOP for reading or writing	FNC_GLOP_Lock
unlock a previously locked GLOP	FNC_GLOP_Unlock
copy the GLOP data in one mapping instance to the GLOP data of all mapping instances on all vprocs on all nodes	FNC_GLOP_Global_Copy
get information as to the specific node ID, vproc ID, vproc type, and task ID for the currently running external routine	FNC_Where_Am_I

## JSON Data Type Interface

The following functions enable a UDF, UDM, or external stored procedure to access and set the value of a JSON parameter, or to get information about the JSON type parameter.

IF you want to ...	THEN use this library function ...
get the maximum length and character set of a JSON type and an indication of whether or not the JSON data is stored as a LOB	FNC_GetJSONInfo
get the maximum length, inline length, character set, storage format of a JSON type instance, and an indication of whether or not the JSON data is stored as a LOB	FNC_GetExtendedJSONInfo
get the value of a JSON type instance when the JSON data is <i>not</i> stored as a LOB	FNC_GetInternalValue
set the value of a JSON type instance when the JSON data will <i>not</i> be stored as a LOB	FNC_SetInternalValue

IF you want to ...	THEN use this library function ...
get a LOB_LOCATOR for a JSON instance which has its data stored as a LOB, and use this locator with LOB FNC routines to read data from the JSON instance	FNC_GetJSONInputLob
get a LOB_RESULT_LOCATOR to a LOB associated with a JSON instance, and use this locator with LOB FNC routines to write data to the JSON instance	FNC_GetJSONResultLob

You can specify the JSON data type as an attribute of a structured UDT. When passing a structured UDT that includes a JSON attribute to or from an external routine, you can use the following interface functions to access or set the value of the JSON attribute, or get information about a JSON attribute.

IF you want to ...	THEN use this library function ...
get information about a JSON attribute, such as the maximum length, character set, or whether or not the data of the attribute is stored as a LOB	FNC_GetStructuredAttributeInfo_EON
get a string representation of a JSON attribute when the data of the JSON attribute is <i>not</i> stored as a LOB	FNC_GetStructuredAttribute
set the string representation of a JSON document to a JSON attribute when the JSON data will <i>not</i> be stored as a LOB	FNC_SetStructuredAttribute
get a LOB_LOCATOR to the LOB which stores the data of a JSON attribute and use this locator with LOB FNC routines to read data from the attribute	FNC_GetStructuredInputLobAttribute
get a LOB_RESULT_LOCATOR to a LOB where the data of a JSON attribute may be stored and use this locator with LOB FNC routines to write data to the attribute	FNC_GetStructuredResultLobAttribute

## LOB Access

LOB access functions enable a UDF, UDM, or external stored procedure to use a locator to access the contents of a referenced object and to append data to a LOB object.

IF you want to ...	THEN use this library function ...
get the length, in bytes, of a large object	FNC_GetLobLength
append a sequence of bytes to a large object that is contained in a UDT or defined to be the result object of a UDF or external procedure	FNC_LobAppend
release all resources associated with a read context, regardless of whether the end of data was reached	FNC_LobClose
convert a locator into a persistent object reference	FNC_LobLoc2Ref

IF you want to ...	THEN use this library function ...
establish a read context for subsequent sequential reads of a referenced object	FNC_LobOpen
perform a sequential read using a specified context	FNC_LobRead
convert a persistent object reference into a locator	FNC_LobRef2Loc

## Memory Allocation

To prevent memory leaks in the database, the `sqltypes_td.h` header file redefines *malloc* and *free* to call the Teradata library functions `FNC_malloc` and `FNC_free`.

## Period Data Type Interface

To access or set the value of a Period parameter or return type, an external routine must use specific library functions.

IF you want to ...	THEN use this library function ...
get the value of a Period type	FNC_GetInternalValue
set the value of a Period type	FNC_SetInternalValue

## Query Band Access

A UDF, UDM, or external stored procedure can use query band access functions to retrieve query band name-value pairs that have been set on a session, transaction, or profile to identify the originating source of queries and help manage task priorities and track system use.

IF you want to ...	THEN use this library function ...
retrieve the current query band string for the transaction, session, and profile	FNC_GetQueryBand
retrieve transaction, session, or profile name-value pairs from the query band string that <code>FNC_GetQueryBand</code> returns	FNC_GetQueryBandPairs
search the transaction, session, and/or profile name-value pairs in the query band string that <code>FNC_GetQueryBand</code> returns and retrieve the value for a specified name	FNC_GetQueryBandValue

## Stored Procedure Invocation

External stored procedures can call the stored procedure invocation function.

IF you want to ...	THEN use this library function ...
call stored procedures from within an external stored procedure	FNC_CallSP

## String Argument and Result Processing

A UDF, UDM, or external stored procedure can use string argument and result processing functions when working with BYTE, CHAR, CHARACTER(n), CHARACTER SET GRAPHIC, or VARCHAR input parameters, output parameters, or results.

These functions are particularly useful for UDFs that are used for the algorithmic compression and decompression of table columns.

IF you want to ...	THEN use this library function ...
determine the size of the output buffer that a scalar or aggregate UDF must return as the result	FNC_GetOutputBufferSize
get the length, in bytes, for an external routine input argument that has a data type of BYTE	FNC_GetByteLength
get the length, in bytes, for an external routine input argument that has a data type of CHAR	FNC_GetCharLength
get the length, in bytes, for an external routine input argument that has a data type of CHARACTER CHARACTER SET GRAPHIC	FNC_GetGraphicLength
get the length, in bytes, for an external routine input argument that has a data type of VARCHAR	FNC_GetVarCharLength
set the length, in bytes, for an external stored procedure output parameter or the result of a UDF or UDM that has a VARCHAR data type	FNC_SetVarCharLength

## Table Function Processing

The table function processing functions are called by a table UDF.

IF you want to ...	THEN use this library function ...
determine how the table function was called in the FROM TABLE clause of the SELECT statement and what action to perform	FNC_GetPhase
	FNC_GetPhaseEx
designate one copy of the table function to be a controlling copy that can distribute global control data among other copies running on other AMP vprocs	FNC_TblControl
use a control scratchpad to propagate data from the table function control copy to all other copies running on all other AMP vprocs	FNC_TblAllocCtrlCtx
	FNC_TblGetCtrlCtx

IF you want to ...	THEN use this library function ...
use a general scratchpad to retain data between iterations of a local table function copy	FNC_TblAllocCtx
	FNC_TblGetCtx
not participate in the process of returning rows	FNC_TblOptOut
gracefully abort a request when a copy of the table function encounters an error condition and cannot continue	FNC_TblAbort
obtain node ID and AMP ID information that allows table functions to configure themselves to run on specific AMPs	FNC_TblGetNodeData
	FNC_AMPInfo
implement a table function that can run on any AMP and only needs one copy to participate in the transaction and request	FNC_TblFirstParticipant
get the definitions of the result columns that must be returned by a table function with dynamic result row specification	FNC_TblGetColDef

## Table Operator Interface

The table operator interface functions allow table operator and contract function writers to access and set metadata. They also provide an interface to read and write rows in input and output streams.

### Note:

To run in nonprotected mode, C language table operators must be thread safe since the AMP environment is multithreaded.

The table operator FNC functions handle character sets as follows:

- The character set of any string parameter will match the character set defined when the table operator was created. This is the same as existing UDF behavior.
- Any functions (such as FNC\_TblOpGetCustomValue) that perform string comparisons will be case insensitive.

The following functions provide access to the metadata associated with the entire operator or with a specific stream. They also provide mechanisms to set metadata for the entire operator or for an individual stream.

IF you want to ...	THEN use this library function ...
retrieve the unique identifier associated with a table operator	FNC_TblOpGetUniqID
retrieve column definitions of a stream	FNC_TblOpGetColDef
retrieve the number of columns in a stream	FNC_TblOpGetColCount
get the information on the base type or attribute types for a UDT or complex data type (CDT)	FNC_TblOpGetBaseInfo



IF you want to ...	THEN use this library function ...
retrieve metadata information about one or more UDT columns for an input or output stream	FNC_TblOpGetUDTMetadata
retrieve information on all of the attributes of a structured UDT type	FNC_TblOpGetStructuredAttributeInfo
retrieve the number of Custom clause keys	FNC_TblOpGetCustomKeyCount
retrieve the number of values associated with a key in a Custom clause, their total size in bytes, and their type	FNC_TblOpGetCustomKeyInfoOf
retrieve the number of values in a Custom clause at a given index, their total size in bytes, and their type	FNC_TblOpGetCustomKeyInfoAt
retrieve all the values associated with a key in a Custom clause	FNC_TblOpGetCustomValuesOf
retrieve the alias name (AS name) associated with an input stream	FNC_TblOpGetAsClauseName
retrieve the HASH BY information for an input stream	FNC_TblOpGetHashByDef
retrieve the number of columns in the HASH BY clause	FNC_TblOpGetCountHashByDef
retrieve the LOCAL ORDER BY information for an input stream	FNC_TblOpGetLocalOrderByDef
retrieve the number of columns in the LOCAL ORDER BY clause	FNC_TblOpGetCountLocalOrderByDef
allow the contract function writer to set the HASH BY specification	FNC_TblOpSetHashByDef
allow the contract function writer to set the ordering specifications	FNC_TblOpSetLocalOrderByDef
find out whether or not the input to the table operator is DIMENSION input	FNC_TblOpIsDimension
communicate the output columns of an output stream to the parser	FNC_TblOpSetOutputColDef
set an opaque binary string value (the contract function context) that the contract function passes to the associated table operator at execution time.	FNC_TblOpSetContractDef
retrieve the length of the contract function context	FNC_TblOpGetContractLength
retrieve the contract function context	FNC_TblOpGetContractDef
set the format of an input or output stream	FNC_TblOpSetFormat
get the default format or the format set (by FNC_TblOpSetFormat) in the contract function	FNC_TblOpGetFormat
generate an error message	FNC_TblOpSetError
access information about the calling table operator	FNC_TblOpGetFunctionDef

These functions are LOB-related interface functions.

IF you want to ...	THEN use this library function ...
establish a read context for subsequent sequential reads of a referenced object	FNC_LobOpen_CL
convert an output column index into a LOB_RESULT_LOCATOR, which can then be used by FNC_LOBAppend	FNC_LobCol2Loc
retrieve the length of an input LOB	FNC_GetLobLength_CL

The following functions allow table operator writers to open streams, read rows, write rows, and close streams. In addition, you can get and set values of specific attributes in the current row of a stream.

IF you want to ...	THEN use this library function ...
get the number of input and output streams passed to the table operator	FNC_TbIOpGetStreamCount
initialize the iterator interface for reading or writing a stream	FNC_TbIOpOpen
read rows from an input stream and set the read context to the next input row of data	FNC_TbIOpRead
access a specific input attribute value	FNC_TbIOpGetAttributeByNdx
bind an output attribute value to a memory location	FNC_TbIOpBindAttributeByNdx
write current output data to spool and set output context to the next output row	FNC_TbIOpWrite
close a stream and flush the data to the database	FNC_TbIOpClose

These functions provide access to input or output buffers.

IF you want ...	THEN use this library function ...
high performance direct buffer read access to the current input buffer	FNC_TbIOpReadBuf
high performance direct buffer read access with support for multiple input streams	FNC_TbIOpReadBufEx
high performance direct buffer write access	FNC_TbIOpWriteBuf

The following function disables the cogroup functionality for table operators that handle multiple inputs streams.

IF you want ...	THEN use this library function ...
to turn off cogroup functionality	FNC_TbIOpDisableCoGroup

The following functions can be used by table operators to import and export data from and to foreign servers.

IF you want to ...	THEN use this library function ...
get values that hash to the specified AMPs	FNC_GetAmpHash
determine the AMP which would be responsible for a key based on the input	FNC_GetHashAmp
set the number of rows exported	FNC_SetActivityCount
obtain node ID and AMP ID information that allows table functions and table operators to configure themselves to run on specific AMPs	FNC_TblOpGetNodeData
record the number of bytes transferred between Vantage and the foreign server by the table operator	FNC_TblOpBytesTransferred
get the information on the base type or attribute types for a UDT or complex data type (CDT)	FNC_TblOpGetBaseInfo
retrieve column definitions of a stream and get the output column definition for the contract function	FNC_TblOpGetColDef
retrieve the contract function context	FNC_TblOpGetContractDef
get the phase in the parser from which the contract function is being called	FNC_TblOpGetContractPhase
get the text query string for the foreign server and get the interface version that is currently supported	FNC_TblOpGetExternalQuery
get the contract definition of a nested inner table operator for the outer table operator to use	FNC_TblOpGetInnerContract
set an opaque binary string value (the contract function context) that the contract function passes to the associated table operator at execution time	FNC_TblOpSetContractDef
reset the lengths in column definitions for VARCHAR data types	FNC_TblOpSetDisplayLength
set the EXPLAIN text when the table operator has the hexplain custom clause set	FNC_TblOpSetExplainText
set attributes of the format of the input and output streams	FNC_TblOpSetFormat
allow the contract function writer to set the HASH BY specification	FNC_TblOpSetHashByDef
set casting statements on the input columns so that the data types are cast as indicated by the caller	FNC_TblOpSetInputColTypes
allow the contract function writer to set the ordering specification when developing table operators	FNC_TblOpSetLocalOrderByDef

For details about the data structures used by the table operator FNC functions, see [Table Operator Data Structures](#).

## TD\_ANYTYPE Parameter Access

The TD\_ANYTYPE parameter access function enables a UDF, UDM, or external stored procedure to retrieve information about TD\_ANYTYPE input and output parameters.

IF you want to ...	THEN use this library function ...
get information about the TD_ANYTYPE arguments passed into a routine	FNC_GetAnyTypeInfo_eon

## Trace

The trace library functions let you get trace output for debugging purposes during UDF, UDM, and external stored procedure development.

IF you want to ...	THEN use this library function ...
retrieve the function trace string specified in the SET SESSION FUNCTION TRACE statement	FNC_Trace_String
write trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement	FNC_Trace_Write_DL

## UDT Interface

UDT interface functions enable a UDF, UDM, or external stored procedure to access and set the value of a distinct UDT or attribute values of a structured UDT.

IF you want to ...	THEN use this library function ...
get the value of a distinct type	FNC_GetDistinctValue
set the value of a distinct type	FNC_SetDistinctValue
get the locator for a distinct type that represents a LOB	FNC_GetDistinctInputLob
	FNC_GetDistinctResultLob
get the number of attributes of a structured type	FNC_GetStructuredAttributeCount
get information, such as data type, about the attributes of a structured type	FNC_GetStructuredAttributeInfo_EON
get the attribute value of a structured type	FNC_GetStructuredAttribute
	FNC_GetStructuredAttributeByNdx
set the attribute value of a structured type	FNC_SetStructuredAttribute
	FNC_SetStructuredAttributeByNdx
get the locator for a structured type LOB attribute	FNC_GetStructuredInputLobAttribute

IF you want to ...	THEN use this library function ...
	FNC_GetStructuredInputLobAttributeByNdx
	FNC_GetStructuredResultLobAttribute
	FNC_GetStructuredResultLobAttributeByNdx

## UDT Serialization

These functions are used to serialize/deserialize UDT parameters and return values for UDTs that support serialization and deserialization.

IF you want to ...	THEN use this library function ...
determine whether a UDT supports serialization and deserialization	FNC_UdtSerializeSupported
retrieve the actual size of a UDT in its serialized form	FNC_UdtGetSerializeSize
retrieve a UDT in its serialized format	FNC_UdtSerialize
deserialize data in a UDT's serialized format back into the UDT	FNC_UdtDeserialize

## XML Data Type Interface

The following functions enable a UDF, UDM, or external stored procedure to access and set the value of an XML parameter, or to get information about the XML type parameter.

IF you want to ...	THEN use this library function ...
get information about the XML value including its size and whether it stores its value as a LOB	FNC_GetXMLInfo
get the value of an XML type instance when the XML data is <i>not</i> stored as a LOB	FNC_GetXML
get a LOB locator for the CLOB representation of the XML type, and use this locator with LOB FNC routines to read data	FNC_GetXMLClob
get a LOB_RESULT_LOCATOR that will be used with LOB FNC routines to set the CLOB value ( the XML return value)	FNC_GetXMLResultClob
set the value of an XML type instance when the XML data will <i>not</i> be stored as a LOB	FNC_SetXML
set the XML return value or OUT parameter value using a CLOB locator returned by FNC_GetXMLResultClob	FNC_SetXMLClob
get the value of an XML type in UTF-8 binary encoding when the XML data is <i>not</i> stored as a LOB	FNC_GetXMLByte
get a LOB locator for the BLOB representation (in UTF-8 encoding) of the XML type, and use this locator with LOB FNC routines to read data	FNC_GetXMLBlob

IF you want to ...	THEN use this library function ...
set the value of an XML type instance (using a value in UTF-8 encoding) when the XML data will <i>not</i> be stored as a LOB	FNC_SetXMLByte
get a LOB_RESULT_LOCATOR that will be used with LOB FNC routines to set the BLOB value ( the XML return value)	FNC_GetXMLResultBlob
set the XML return value or OUT parameter value using a BLOB locator returned by FNC_GetXMLResultBlob	FNC_SetXMLBlob

## FNC Data Structures

### DATASET Definitions

This section describes the data structures defined in `sqltypes_td.h` for use with external routines that access or manipulate DATASET data.

#### DATASET\_HANDLE

A DATASET argument is passed to an external routine using a DATASET\_HANDLE. The DATASET\_HANDLE used to pass DATASET data is defined in `sqltypes_td.h` as follows:

```
typedef int DATASET_HANDLE;
```

#### dataset\_schema\_encoding\_en

Some external routines require a specification of the encoding of the schema text being handled. The `dataset_schema_encoding_en` enum includes the following information. See `sqltypes_td.h` for the complete definition.

- `datasetSchemaUTF8 = 0`
- `datasetSchemaUTF16 = 1`

`sqltypes_td.h` also includes the following definition:

```
typedef Byte dataset_schema_encoding_t;
```

#### dataset\_storage\_en

The `dataset_storage_en` enum specifies the storage format of a DATASET type. It includes the following values. See `sqltypes_td.h` for the complete definition.

- `DATASET_INVALID_EN=-1`
- `DATASET_Avro_EN=0`
- `DATASET_CSV_EN=1`

sqltypes\_td.h also includes the following definition:

```
typedef BYTE dataset_storage_et;
```

## Table Operator Data Structures

This section describes the data structures used by the FNC functions provided to table operator and contract function writers. These data structures are used to store metadata associated with the entire table operator or a specific stream and to pass information to and from the FNC functions.

For details about these data structures, see the sqltypes\_td.h header file.

### FNC\_TblOpColumnDef\_t

Column definitions are associated with streams. This metadata is represented in the data structure FNC\_TblOpColumnDef\_t as a sequence of column types, the length of this sequence, and the number of columns. The size of this data structure is variable.

### parm\_tx

Column types are represented by the data structure parm\_tx.

The following lists some of the information included in this structure. See sqltypes\_td.h for the complete definition.

- The data type of the attribute.
- The name of the column.
- The name of the UDT.
- The JSON storage format.
- The character set.
- The granularity of the period type.
- The maximum size of the fixed-length fields.
- The length in CHAR, VARCHAR, or BYTE types.
- The range for interval types.
- The precision for TIME/TIMESTAMP types.
- $n$  and  $m$  in DECIMAL( $n$ ,  $m$ )

Column names and UDT names are represented as a sequence of up to 128 characters in UNICODE, LATIN, or Kanji.

### dtype\_en

The dtype\_en enum defines all the available data types which may be passed in as a column to a table operator.

The following lists some sample values. See sqltypes\_td.h for the complete definition.

- CHAR\_DT=1
- VARBYTE\_DT=4
- SMALLINT\_DT=8

- REAL\_DT=10
- DECIMAL8\_DT=14
- DATE\_DT=15
- INTERVAL\_MONTH\_DT=20
- TIMESTAMP\_WTZ\_DT=32
- CLOB\_REFERENCE\_DT=34
- UDT\_DT = 35
- NUMBER\_DT=38
- PERIOD\_DT = 39
- JSON\_DT = 40
- DATASET\_AVRO\_DT = 41
- ST\_GEOMETRY\_DT = 42
- MBR\_DT = 43
- MBB\_DT = 44
- ARRAY\_DT = 45
- XML\_DT = 46
- DATASET\_CSV\_DT=47

### **json\_storage\_en**

The `json_storage_en` enum defines the storage format for a JSON type.

The following lists some valid values. See `sqltypes_td.h` for the complete definition.

- JSON\_INVALID\_EN=-1
- JSON\_TEXT\_EN=0
- JSON\_BSON\_EN=1
- JSON\_UBJSON\_EN=2

### **period\_en**

The granularity of Period types is defined by the `period_en` enum.

The following lists some valid values. See `sqltypes_td.h` for the complete definition.

- NOT\_PERIOD = 0
- PERIOD\_DATE = 1
- PERIOD\_TIME = 2
- PERIOD\_TIME\_WTZ = 3
- PERIOD\_TIMESTAMP = 4,
- PERIOD\_TIMESTAMP\_WTZ = 5



**FNC\_TbLOpHandle\_t**

Handles are structures used to access streams and can be used to pass information to and from FNC functions. Functions that access metadata associated with streams require handles as one of their input parameters. However, functions that access metadata associated with the operator do not require a handle.

Handles are represented by the data structure `FNC_TbLOpHandle_t`.

The following lists some of the information included in this structure. See `sqltypes_td.h` for the complete definition.

- The stream number.
- A pointer to information about the current row in the stream.  
If attributes in the current row can be directly accessed using information stored in this field, then the `Options` field is set to 0.
- The direction of the stream (input or output).
- Options (whether attributes in a row can be accessed directly or not).
- State (whether the stream has been opened or closed).

**Stream\_Direction\_en**

The direction of a stream is defined by the `Stream_Direction_en` enum.

Valid values include the following. See `sqltypes_td.h` for the complete definition.

- `ISOUTPUT = 'W'`
- `ISINPUT = 'R'`

**Stream\_State\_en**

The state of a stream is defined by the `Stream_State_en` enum.

Valid values include the following. See `sqltypes_td.h` for the complete definition.

- `ISINIT = 1`
- `ISOPEN = 2`
- `ISCLOSE = 3`

**current\_row\_t**

Information about the current row in a stream is stored in the data structure `current_row_t`. This includes locations of individual attributes (`columnptr`) where you can have direct access to them.

The following lists some of the information included in this structure. See `sqltypes_td.h` for the complete definition.

- The length of the row data in bytes.
- The record type of the body (matches parcel flavors).
- The location of the indicators. A pointer to NULL indicators for fields in the current record.
- The location of the column data.

- Lengths for column data.
- A pointer to the raw row record body.

### **FNC\_Names\_t and FNC\_Names\_Ord\_t**

FNC functions that handle HASH BY and LOCAL ORDER BY metadata have sequences of column names as input or output parameters. The data structure `FNC_Names_t` stores these sequences and `FNC_Names_Ord_t` stores sequences of names with an order (ascending or descending).

See `sqltypes_td.h` for the definitions of these structures.

### **Key\_info\_t**

Functions that retrieve values associated with a key in a custom clause use the data structure `Key_info_t` to store these values.

The following lists some of the information included in this structure. See `sqltypes_td.h` for the complete definition.

- The number of values.
- The total size of the values in bytes.
- The size of the key.
- The data type of the values.
- An array of values.
- The key.

### **UDT\_Baseinfo\_t**

The `UDT_BaseInfo_t` structure provides metadata about a UDT/CDT input or output column. This structure is used only for table operators.

The following lists some of the information included in this structure. See `sqltypes_td.h` for the complete definition.

- The type of the UDT.
- If it is an ARRAY UDT, the dimensions of the array, the number of elements, and information about each element.
- Information about the base type of the UDT or CDT, such as the character set, the data type, and so forth.
- For JSON types, the JSON storage format.
- For DATASET types, the DATASET storage format.
- The number of attributes for a structured UDT.
- The name of the UDT.
- The predefined data type which the UDT is mapped from and to for transforms.

The information about the base type of the UDT or CDT is specified in the `base_*` fields of the `UDT_BaseInfo_t` structure. The following table shows the base data type mapping for a UDT or CDT.

SQL UDT or CDT	Attribute/Element/Base Type Code
Distinct UDT	Predefined data type the UDT is based on. For example, if you have <code>CREATE TYPE myint as INTEGER FINAL;</code> Then the Base type code is <code>INTEGER_DT</code> .
Structured UDT	Since a structured UDT may have many attributes and may be nested, the attributes are not saved in the metadata but are accessed by calling the <code>FNC_TblOpGetStructuredAttributeInfo</code> function.
Period types	Date/Time base type of <code>BEGIN</code> and <code>END</code> elements. Valid values are <code>DATE_DT</code> , <code>TIME_DT</code> , <code>TIMESTAMP_DT</code> , <code>TIME_WTZ_DT</code> , <code>TIMESTAMP_WTZ_DT</code> .
XML	<code>CLOB_REFERENCE_DT</code>
Geospatial – ST_Geometry	<code>CLOB_REFERENCE_DT</code>
Geospatial – MBR	<code>VARCHAR_DT</code>
Geospatial – MBB	<code>VARCHAR_DT</code>
ARRAY/VARRAY	The data type of the element of the array. For example, if you have <code>CREATE TYPE intary as INTEGER ARRAY[10];</code> Then the type code of the element is <code>INTEGER_DT</code> .
JSON	<code>CLOB_REFERENCE_DT</code>
BSON	<code>BLOB_REFERENCE_DT</code> or <code>CLOB_REFERENCE_DT</code>
DATASET	<code>BLOB_REFERENCE_DT</code> or <code>CLOB_REFERENCE_DT</code>

Note that the `base_*` fields of the `UDT_BaseInfo_t` structure are not filled in for structured UDTs. Since structured UDTs may have many attributes and may also contain an arbitrary level of nesting, metadata about the attributes of a structured UDT is retrieved using the `FNC_TblOpGetStructuredAttributeInfo` function. `FNC_TblOpGetStructuredAttributeInfo` returns an array of `attribute_info_t` structures corresponding to all of the attributes in the structured UDT.

#### **attribute\_info\_t and attribute\_info\_eon\_t**

The `attribute_info_t` and `attribute_info_eon_t` data structures describe an attribute of a structured UDT.

The following lists some of the information included in these structures. See `sqltypes_td.h` for the complete definition.

- The attribute positional index.
- The attribute data type.
- The attribute name.
- The UDT type indicator.
- The UDT type name.
- For JSON types, the JSON storage format.
- The maximum length for this data type.
- The LOB length for LOB data types.
- The character set.

### **SMALLINT** `udt_indicator`

Indicates the type of UDT or CDT.

The following lists some sample values. See `sqltypes_td.h` for the complete definition.

- 0=Not a UDT or CDT
- 1=Array
- 2=Structured
- 3=JSON ENCODE AS TEXT
- 4=Distinct
- 5=Period types
- 6=XML
- 7=ST\_Geometry
- 8=MBR
- 9=MBB
- 10=DATASET STORAGE FORMAT AVRO
- 11=DATASET STORAGE FORMAT CSV

## **FNC\_AmpInfo**

Returns AMP-specific information that a local copy of a table function or table operator can use to configure itself to use the correct resources.

## **Return Type**

`FNC_AMPInfo` returns a pointer to an `AMP_Info_t` structure, defined as follows:

```
typedef struct {
    unsigned short NodeId;
```

```

    unsigned short AMPId;
    unsigned short LowestAMPOnNode;
} AMP_Info_t;

```

***NodeId***

Unique number of the node.

***AMPId***

Unique number of the AMP.

***LowestAMPOnNode***

Whether the current AMP is the lowest AMP on the same node as the invoking AMP: Zero if yes, nonzero if no.

If this function is invoked from a table operator that is associated with a map, then *LowestAMPOnNode* specifies whether the current AMP is the lowest AMP on the same node within the specified map.

A table function or operator that allows only the lowest AMP vproc to participate can use this field.

## Syntax

```

AMP_Info_t *
FNC_AMPInfo(void);

```

## Usage Notes

A table function or table operator that is implemented to run local copies on specific AMP vprocs can use this function with `FNC_TblGetNodeData` to determine whether a local copy should participate in the table function processing or opt out.

## Restrictions

This function can only be called from within a table function or table operator. Calling this function from a scalar or aggregate function results in an exception on the transaction.

## Example

```

AMP_Info_t *LocalConfig;

LocalConfig = FNC_AMPInfo();

```

## FNC\_CallSP

Provides a way to call a stored procedure from within an external stored procedure.

### Syntax

```
void *
FNC_CallSP( SQL_TEXT *SP_Name,
            int      argc,
            void     *argv[],
            int      ind[],
            parm_t    dtype[],
            char      *sqlstate )
```

### Syntax Elements

#### *SP\_Name*

Name of the stored procedure to invoke:

- The name can be qualified by the name of the database in which the stored procedure resides.
- The name can be up to 128 characters.

#### *argc*

Count of arguments to pass to the stored procedure. The value must match the number of parameters expected by the stored procedure being called.

#### *argv*

Entry points to the argument values to be passed or returned from the stored procedure being called. The number of elements in *argv* must match the number of parameters expected by the stored procedure being called.

#### *ind*

Indicators corresponding to the arguments of the stored procedure being called, in the same order.

If the *argv* element is an IN or INOUT argument, and the value is:

- null, then set the corresponding element in *ind* to -1.
- not null, then set the corresponding element in *ind* to 0.

For elements of *argv* that are OUT or INOUT arguments of the stored procedure being called, the value of the corresponding element in *ind* will be returned by the stored procedure being called, according to the preceding rules.

### ***dtype***

Data type, attributes, and direction (IN, OUT, or INOUT) of each argument being passed to or returned from the stored procedure.

The `parm_t` structure is defined in `sqltypes_td.h` as:

```
typedef struct parm_t
{
    dtype_et    datatype;
    dmode_et    direction;
    charset_et  charset;
    union {
        long    length;
        int     intervalrange;
        int     precision;
        struct {
            int  totaldigit;
            int  fracdigit;
        } range;
    } size;
} parm_t;
```

where:

- `datatype` specifies the data type of the argument. The `sqltypes_td.h` header file defines `dtype_et` as:

```
typedef int dtype_et;
```

Valid values are defined by the `dtype_en` enumeration in `sqltypes_td.h`:

```
typedef enum dtype_en {
    UNDEF_DT=0,
    CHAR_DT=1,
    VARCHAR_DT=2,
    CLOB_REFERENCE_DT=34,
    BYTE_DT=3,
    VARBYTE_DT=4,
    BLOB_REFERENCE_DT=33,
    GRAPHIC_DT=5,
    VARGRAPHIC_DT=6,
```

```

    BYTEINT_DT=7,
    SMALLINT_DT=8,
    INTEGER_DT=9,
    BIGINT_DT=36,
    REAL_DT=10,
    DECIMAL1_DT=11,
    DECIMAL2_DT=12,
    DECIMAL4_DT=13,
    DECIMAL8_DT=14,
    DECIMAL16_DT=37,
    NUMBER_DT=38,
    DATE_DT=15,
    TIME_DT=16,
    TIMESTAMP_DT=17,
    INTERVAL_YEAR_DT=18,
    INTERVAL_YTM_DT=19,
    INTERVAL_MONTH_DT=20,
    INTERVAL_DAY_DT=21,
    INTERVAL_DTH_DT=22,
    INTERVAL_DTM_DT=23,
    INTERVAL_DTS_DT=24,
    INTERVAL_HOUR_DT=25,
    INTERVAL_HTM_DT=26,
    INTERVAL_HTS_DT=27,
    INTERVAL_MINUTE_DT=28,
    INTERVAL_MTS_DT=29,
    INTERVAL_SECOND_DT=30,
    TIME_WTZ_DT=31,
    TIMESTAMP_WTZ_DT=32,
    UDT_DT=35,
    FNC_DATATYPESETSIZE=39
} dtype_en;

```

- direction specifies the direction of the argument (IN, OUT, or INOUT).

The `sqltypes_td.h` header file defines `dmode_et` as:

```
typedef int dmode_et;
```

Valid values are defined by the `dmode_en` enumeration in `sqltypes_td.h`:

```

typedef enum dmode_en
{
    UNDEF_PM=0,

```



```

    IN_PM=1,
    INOUT_PM=2,
    OUT_PM=3
} dmode_en;

```

- charset specifies the character set of CHAR or VARCHAR data.

The `sqltypes_td.h` header file defines `charset_et` as:

```
typedef int charset_et;
```

Valid values are defined by the `charset_en` enumeration in `sqltypes_td.h`:

```

typedef enum charset_en
{
    UNDEF_CT=0,
    LATIN_CT=1,
    UNICODE_CT=2,
    KANJISJIS_CT=3,
    KANJI1_CT=4
} charset_en;

```

- `size.length` specifies the length of a CHAR, VARCHAR, BYTE, or CHARACTER CHARACTER SET GRAPHIC type.
- `size.intervalrange` specifies the range of an INTERVAL type. For example, 4 for INTERVAL YEAR(4).
- `size.precision` specifies the precision in a TIME or TIMESTAMP type. For example, 4 for TIME(4).
- `size.range.totaldigit` specifies the value *m* in a DECIMAL(*m*,*n*) or INTERVAL SECOND (*m*,*n*) type.
- `size.range.fracdigit` specifies the value *n* in a DECIMAL(*m*,*n*) or INTERVAL SECOND (*m*,*n*) type.

### ***sqlstate***

a pointer to a `char[6]` array that is used to return the result of executing the called stored procedure.

## **Usage Notes**

The following notes apply to the usage of `FNC_CallSP`:

- When making nested stored procedure calls, only one of those procedures can be an external procedure. For example, an external stored procedure cannot call a stored procedure that in turn calls an external stored procedure.

- To call a stored procedure from an external stored procedure that uses CLIV2 to execute SQL, the best practice is to submit a CALL statement using CLIV2 instead of using FNC\_CallSP.
- Normally, a stored procedure can only execute statements corresponding to the access clause of the most restrictive procedure that called it. For example, consider the following:
  - Stored procedure sp1 where the CREATE PROCEDURE statement specifies a data access clause of NO SQL (the default).
  - Stored procedure sp2 where the CREATE PROCEDURE statement specifies a data access clause of MODIFIES SQL DATA.

If stored procedure sp1 calls stored procedure sp2, stored procedure sp2 cannot execute SQL statements because the caller is already restricted by the NO SQL data access clause.

However, if you use FNC\_CallSP to call stored procedure sp2 from an external stored procedure, the data access clause of the external stored procedure is ignored and stored procedure sp2 can execute SQL statements that modify data.

## Restrictions

This function can only be called from within an external stored procedure.

None of the IN, INOUT, or OUT arguments of the stored procedure that this function calls can have a data type of BLOB or CLOB.

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xsp_getregion ( VARCHAR_LATIN *region,
                    char           sqlstate[6])
{
    void *argv[2];
    int   ind[2];
    parm_t dtype[2];
    INTEGER regionCount; /* OUT argument from addRegion */

    /* Set the pointers to the stored procedure arguments */
    argv[0] = region;      /* IN */
    argv[1] = &regionCount; /* OUT */
}
```

```

/* Set the indicator for the IN argument */
ind[0] = 0;
memset(dtype, 2, sizeof(parm_t)*2);

/* Data type for the VARCHAR IN argument */
dtype[0].datatype = VARCHAR_DT;
dtype[0].direction = IN_PM;
dtype[0].charset = LATIN_CT;
dtype[0].size.length = strlen((const char *)region);

/* Data type for the INTEGER OUT argument */
dtype[1].datatype = INTEGER_DT;
dtype[1].direction = OUT_PM;
FNC_CallSP((SQL_TEXT *)"addRegion", 2, argv, ind, dtype, sqlstate);

if (strcmp(sqlstate, "00000") != 0)
    return;

...
}

```

## FNC\_CheckNullBitVector

Checks the value of one bit in a NullBitVector that was previously allocated by the caller.

FNC\_CheckNullBitVector returns an integer which is the value of the bit specified by *indexValue*. Valid values returned are:

- 1, meaning that the corresponding element in the ARRAY contains a non-null value.
- 0, meaning that the corresponding element in the ARRAY is present but set to NULL.

## Syntax

```

int
FNC_CheckNullBitVector ( NullBitVecType *NullBitVector,
                        int             indexValue,
                        long            bufSize)

```

### Syntax Elements

#### **NullBitVector**

a NullBitVector array previously allocated by the caller.

The data type used to access the NullBitVector is defined in `sqltypes_td.h` as:

```
typedef unsigned char NullBitVecType;
```

***indexValue***

the bit in the NullBitVector to be checked, as specified in row-major order for an ARRAY. Valid values start from 0.

***bufSize***

the size in bytes of the NullBitVector as allocated by the caller prior to initialization of the NullBitVector by setting all bytes to 0.

## Usage Notes

FNC\_CheckNullBitVector takes *NullBitVector*, *indexValue*, and *bufSize* as input and returns the value of the presence bit specified by *indexValue* in the NullBitVector.

After calling an FNC routine such as FNC\_GetArrayElements, which sets presence information in a NullBitVector, you can call FNC\_CheckNullBitVector to interpret the results of the NullBitVector returned by FNC\_GetArrayElements.

For more information about using NullBitVectors, see [Checking and Setting the NullBitVector](#).

## Example

In this example, FNC\_CheckNullBitVector is called to check the value of element 2 after calling FNC\_GetArrayElements.

```
void VerifyArrayLength ( ARRAY_HANDLE *ary_handle,
                        char          sqlstate[6])
{
    NullBitVecType *NullBitVector;
    array_info_t arrayInfo;
    long nullVecBufSize;
    int presenceBit;
    bounds_t *arrayRange;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];
    index_t lastPresentElement;

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*ary_handle,
                        &arrayInfo,
                        arrayScope);
    (arrayInfo.totalNumElements % 8 == 0) ?
```

```

    (nullVecBufSize = arrayInfo.totalNumElements / 8) :
    (nullVecBufSize = arrayInfo.totalNumElements / 8) + 1;

/* Allocate a new NullBitVector array. */
NullBitVector = (NullBitVecType*)FNC_malloc(nullVecBufSize);

/* Initialize the NullBitVector to default values. */
memset(NullBitVector, 0, nullVecBufSize);
arrayRange =
(bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

/* Set values of arrayRange to correspond to the range [1:2][1:2] */
arrayRange[0].lowerBound = 1;
arrayRange[0].upperBound = 2;
arrayRange[1].lowerBound = 1;
arrayRange[1].upperBound = 2;

/* Get elements within the range [1:2][1:2] of myArray. */
FNC_GetArrayElements(*ary_handle, arrayRange, &result, buffer,
    NullBitVector, &length);

/* Check the presence bit for element [1][1] */
presenceBit = FNC_CheckNullBitVector(NullBitVector, 1,
    nullVecBufSize);
...
}

```

## FNC\_CheckNullBitVectorByElemIndex

Checks the value of one bit in a NullBitVector that was previously allocated by the caller. The bit to be checked may be referenced by the ARRAY type element index as specified by dimension.

FNC\_CheckNullBitVectorByElemIndex returns an integer which is the value of the specified bit that was checked in the NullBitVector. Valid values returned are:

- 1, meaning that the corresponding element in the ARRAY contains a non-null value.
- 0, meaning that the corresponding element in the ARRAY is present but set to NULL.

## Syntax

```

int
FNC_CheckNullBitVectorByElemIndex ( NullBitVecType *NullBitVector,
                                   int             indexValue[],
                                   long            bufSize,

```

```

        bounds_t    *arrayScope,
        int          numDimensions)

```

## Syntax Elements

### ***NullBitVector***

NullBitVector array previously allocated by the caller.

The data type used to access the NullBitVector is defined in `sqltypes_td.h` as:

```
typedef unsigned char NullBitVecType;
```

### ***indexValue***

Index to the ARRAY element whose corresponding presence bit in the NullBitVector is to be checked. For a 1-D ARRAY, the index to the ARRAY element is provided as `indexValue[0]`. If the ARRAY type is n-D, then the complete dimension information for this index is placed in cells `indexValue[1]`, `indexValue[2]`, `indexValue[3]` ... `indexValue[FNC_ARRAYMAXDIMENSIONS]` as needed, where `FNC_ARRAYMAXDIMENSIONS` specifies the maximum number of dimensions in an ARRAY type as defined in `sqltypes_td.h`:

```
#define FNC_ARRAYMAXDIMENSIONS 5
```

### ***bufSize***

Size in bytes of the NullBitVector as allocated by the caller prior to initialization of the NullBitVector by setting all bytes to 0.

### ***arrayScope***

Array of `bounds_t` structures that provides the scope information for the ARRAY which the NullBitVector describes. You can call `FNC_GetArrayTypeInfo` to get this information. See [FNC\\_GetArrayTypeInfo \[Deprecated\]](#).

### ***numDimensions***

Number of dimensions defined for the ARRAY which the NullBitVector describes. You can call `FNC_GetArrayTypeInfo` to get this information. See [FNC\\_GetArrayTypeInfo \[Deprecated\]](#).

## Usage Notes

`FNC_CheckNullBitVectorByElemIndex` takes *NullBitVector*, *indexValue*, *bufSize*, *arrayScope*, and *numDimensions* as input and returns the value of the specified presence bit in the NullBitVector.

After calling an FNC routine such as `FNC_GetArrayElements`, which sets presence information in a `NullBitVector`, you can call `FNC_CheckNullBitVectorByElemIndex` to interpret the results of the `NullBitVector` returned by `FNC_GetArrayElements`.

For more information about using `NullBitVectors`, see [Checking and Setting the NullBitVector](#).

## Example

In this example, `FNC_CheckNullBitVectorByElemIndex` is called to check the value of element 1 of the `NullBitVector`, which corresponds to an `ARRAY` of type `phonenumbers_ary`, after calling `FNC_GetArrayElements`.

```
void ArrayUDF ( ARRAY_HANDLE *ary_handle,
                char          sqlstate[6])
{
    NullBitVecType *NullBitVector;
    array_info_t arrayInfo;
    long nullVecBufSize;
    int presenceBit;
    bounds_t *arrayRange;
    int arrayIndex[FNC_ARRAYMAXDIMENSIONS];
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*ary_handle,
                        &arrayInfo,
                        arrayScope);
    (arrayInfo.totalNumElements % 8 == 0) ?
        (nullVecBufSize = arrayInfo.totalNumElements / 8) :
        (nullVecBufSize = arrayInfo.totalNumElements / 8) + 1;

    /* Allocate a new NullBitVector array. */
    NullBitVector = (NullBitVecType*)FNC_malloc(nullVecBufSize);

    /* Initialize the NullBitVector to default values. */
    memset(NullBitVector, 0, nullVecBufSize);
    arrayRange =
        (bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

    /* Set values of arrayRange to correspond to the range [1:2][1:2] */
    arrayRange[0].lowerBound = 1;
    arrayRange[0].upperBound = 2;
    arrayRange[1].lowerBound = 1;
```

```

arrayRange[1].upperBound = 2;

/* Get elements within the range [1:2][1:2] of myArray. */
FNC_GetArrayElements(*ary_handle, arrayRange, &result, buffer,
    NullBitVector, &length);

/* Check the presence bit for element 1 of phonenumbers_ary. */
arrayIndex[0] = 1;
presenceBit = FNC_CheckNullBitVectorByElemIndex(NullBitVector,
    arrayIndex, nullVecBufSize, arrayScope, arrayInfo.numDimensions);
...
}

```

## FNC\_DbsInfo [Deprecated]

This function is deprecated because it truncates object names to 30 characters. However, it remains available to support legacy applications. For current and future development, use the corresponding function that includes "EON" in the function name. For example, use FNC\_DbsInfo\_EON instead of FNC\_DbsInfo.

Returns user account, user name, user ID, and session number related to the currently running UDF, UDM, table operator, contract function, or external stored procedure.

## Syntax

```

void
FNC_DbsInfo( dbs_info_t *data )

```

**data**

```

typedef struct dbs_info_t {
    SQL_TEXT UserAccount[FNC_MAXNAMELEN];
    SQL_TEXT UserName[FNC_MAXNAMELEN];
    int      UserId;
    short    StatementNo;
    short    Host;
    int      SessionNo;
    int      RequestNo;
} dbs_info_t;

```



## Syntax Elements

### *UserId*

Specifies the user ID.

### *StatementNo*

Specifies the current statement number for the request.

### *Host*

Specifies the host ID.

### *SessionNo*

Specifies the session number.

### *RequestNo*

Specifies the current client request number.

## Usage Notes

You can use FNC\_DbsInfo but it truncates a retrieved User Account Name or User Name after 30 characters. Use FNC\_DbsInfo\_EON instead of FNC\_DbsInfo for longer object names.

## Example

```
dbinfo_t data;

FNC_DbsInfo(&data);
```

## FNC\_DbsInfo\_EON

Returns user account, user name, user ID, and session number related to the currently running UDF, UDM, table operator, contract function, or external stored procedure.

## Syntax

```
void
FNC_DbsInfo_EON( dbinfo_eon_t *data )
```

***data***

```
typedef struct dbs_info_eon_t {
    Unicode_Text UserAccount[FNC_MAXNAMELEN_EON];
    Unicode_Text UserName[FNC_MAXNAMELEN_EON];
    int          UserId;
    short        StatementNo;
    short        Host;
    int          SessionNo;
    int          RequestNo;
} dbs_info_eon_t;
```

## Syntax Elements

***UserId***

Specifies the user ID.

***StatementNo***

Specifies the current statement number for the request.

***Host***

Specifies the host ID.

***SessionNo***

Specifies the session number.

***RequestNo***

Specifies the current client request number.

## Example

```
dbs_info_eon_t data;

FNC_DbsInfo_EON(&data);
```

## FNC\_DBSSessionAttrInfo

Allows table operators to retrieve information about the current session, such as system variables and session attributes.

## Syntax

```
void
FNC_DBSSessionAttrInfo (char  *jsonData);
```

## Syntax Elements

### *jsonData*

Session information returned by the function as a string in JSON format.

## Usage Notes

The current session information returned by FNC\_DBSSessionAttrInfo include the following:

- Current user name
- Current role name
- Transaction mode
- Collation

For current role name, a maximum of 127 role names are returned. If a user is granted more than 127 roles and the user issues a SET ROLE ALL statement to enable all of the roles, "ALL" is returned for the current role name. For example, the following sample output shows the session information returned for TESTUSER who has more than 127 roles enabled:

```
{"CurUserName":"TESTUSER","CurRoleName":"ALL","TransactionMode":"BTET","Collation":"ASCII"}
```

For users who specify SET ROLE ALL but have 127 roles or less, "ALL" is returned along with each of the role names. For example, TESTUSER2 has 3 roles: Role1, Role2, and Role3.

```
{"CurUserName":"TESTUSER2","CurRoleName":
["ALL","Role1","Role2","Role3"],"TransactionMode":"BTET","Collation":"ASCII"}
```

## Example

The following code fragment calls FNC\_DBSSessionAttrInfo to get the current session information string.

```
...
char jsonData [4096] = {'\0'};
FNC_DBSSessionAttrInfo(jsonData);
```

The following sample output shows the current user name, current role name, transaction mode, and collation.

```
{"CurUserName": "TESTUSER", "CurRoleName": "R1", "TransactionMode": "BTET", "Collation"
: "ASCII"}
```

## FNC\_DefMem

Allocates the required memory for the aggregate intermediate storage structure defined by an aggregate UDF.

FNC\_DefMem returns a pointer to the allocated memory.

If FNC\_DefMem cannot allocate the memory, which usually only happens when the value of *len* is too big, a NULL pointer is returned.

## Syntax

```
void *
FNC_DefMem ( int  len )
```

## Syntax Elements

*len*

the size, in bytes, of the aggregate intermediate storage structure defined by an aggregate UDF.

## Usage Notes

Vantage maintains the reserved interim memory for the life of the aggregation operation and automatically releases it when done.

## Restrictions

This function can only be called from within an aggregate UDF. Calling this function from an external stored procedure, scalar UDM, scalar function, or table function results in an error.

The value of *interim\_size* in the CLASS AGGREGATE clause of the CREATE FUNCTION statement determines the maximum size of the structure. If *interim\_size* is not specified, then the limit is 64 bytes. The absolute limit is 64000 bytes.

## Example

```
typedef struct {
    FLOAT n;
    FLOAT x_sq;
```

```

    FLOAT x_sum;
} agr_storage;

...

agr_storage *interim_ptr;

...interim_ptr = FNC_DefMem(sizeof(agr_storage));

```

## FNC\_free

Frees temporary memory that was previously allocated by a call to *FNC\_malloc*.

## Syntax

```

void
FNC_free(void *ptr)

```

## Syntax Elements

*ptr*

a pointer to a block previously allocated by *FNC\_malloc*.

## Usage Notes

*FNC\_free* is the equivalent of *free* for use by UDFs, UDMs, table operator, contract function and external stored procedures.

The `sqltypes_td.h` header file redefines *free* to call *FNC\_free*.

WHEN you ...	THEN ...
develop, test, and debug a UDF, UDM, or external stored procedure standalone	include the <code>malloc.h</code> header file and use the standard <i>malloc</i> and <i>free</i> C library functions.
install your UDF, UDM, or external stored procedure on the server	do not include the <code>malloc.h</code> header file and use the definitions of <i>malloc</i> and <i>free</i> from the <code>sqltypes_td.h</code> header file. Note that the definitions are used when submitting source code to the server, but not when submitting objects.

*FNC\_free* checks to make sure the UDF, UDM, or external stored procedure releases all memory before exiting and gives an exception on the transaction if the UDF, UDM, or external stored procedure does not release all temporary memory it allocated. This is to prevent memory leaks in the database.

If you circumvent the logic to call *malloc* and *free* directly, then there is a good chance that a memory leak could occur even if the UDF, UDM, or external stored procedure frees up memory correctly. The reason is that a UDF, UDM, or external stored procedure can abort while it is running if a user aborts the transaction, or if the transaction aborts because of some constraint violation that might occur on another node unbeknownst to the running UDF, UDM, or external stored procedure.

## FNC\_GeomGetResultWKBBlob

Returns the LOB locator to a Well-Known Binary (WKB) BLOB for an ST\_Geometry that is defined to be the return value of a UDF/UDM or an INOUT or OUT parameter for an external stored procedure.

### Syntax

```
void
FNC_GeomGetResultWKBBlob(GEO_HANDLE      geoHandle,
                          LOB_RESULT_LOCATOR* geoBlob)
```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *geoBlob*

Pointer to the LOB locator that is used to set the ST\_Geometry return value.

### Usage Notes

The FNC\_GeomGetResultWKBBlob function gets a result LOB locator that is used to set the ST\_Geometry return value. The ST\_Geometry Handle *geoHandle* is passed as input and a LOB result locator *geoBlob* is returned from the function. The user can then use LOB FNC routines to set the BLOB value using the *geoBlob* locator. The Well-Known Binary value should be written to the BLOB.

The FNC\_GeomGetResultWKBBlob should be called only for LOB-based ST\_Geometry values. If used with inline ST\_Geometry values, a 7579 error is generated: FNC\_GeomGetResultWKBBlob can only be called on LOB-based ST\_Geometry type.

### **Example: Using FNC\_GeomGetResultWKBBlob and FNC\_GeomSetWKBBlob to retrieve and set the value of a return ST\_Geometry**

The following function takes a ST\_Geometry parameter and returns an ST\_Geometry that has the same value. It first retrieves the Well-Known Binary format from the input geometry. It then retrieves the LOB result locator from the return ST\_Geometry, appends the WKB to it, and then sets the value of the return ST\_Geometry.

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void geo_wkb( GEO_HANDLE  *geo_in,
              GEO_HANDLE  *geo_out,
              int          *indicator_geo_in,
              int          *indicator_geo_out,
              char          sqlstate[6],
              SQL_TEXT     extname[129],
              SQL_TEXT     specific_name[129],
              SQL_TEXT     error_message[257])
{
    LOB_RESULT_LOCATOR geoBlob_out;
    LOB_LOCATOR geoBlob_in;
    BYTE buffer[64000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_LobLength_t blob_length;
    int srid = -1;
    int max_length, max_length2, numLobs, numLobs2;

    /* If input is null set return value to null. */
    if (*indicator_geo_in == -1 )
    {
        *indicator_geo_out = -1;
        return;
    }

    /* Get the Input Geometry Info */
    FNC_GetGeometryInfo(*geo_in,&max_length, &numLobs);

    /* Get the Output Geometry Info */
    FNC_GetGeometryInfo(*geo_out,&max_length2, &numLobs2);

    /*
     * Copy geo_in ST_Geometry value to geo_out ST_Geometry value.
     */
    if(numLobs ==1 && numLobs2 ==1)
    {

```

```

/* Get the LOB_RESULT_LOCATOR of geo_out. */
FNC_GeomGetResultWKBBlob(*geo_out, &geoBlob_out);

/* Get the LOB_LOCATOR and srid of the input ST_Geometry value. */
FNC_GeomGetWKBBlob(*geo_in, &geoBlob_in, &srid);

FNC_LobOpen(geoBlob_in, &id, 0, 0);
blob_length = FNC_GetLobLength(geoBlob_in);
while (FNC_LobRead(id, buffer, blob_length, &actlen) == 0
      && !trunc_err)
    trunc_err = FNC_LobAppend(geoBlob_out, buffer, actlen, &actlen);
FNC_LobClose(id);

/* Set the return ST_Geometry value */
FNC_GeomSetWKBBlob(*geo_out, geoBlob_out, srid);
}
}

```

## FNC\_GeomGetResultWKTClob

Returns the LOB locator to a CLOB for an ST\_Geometry that is defined to be the return value of a UDF/UDM or an INOUT or OUT parameter for an external stored procedure.

### Syntax

```

void
FNC_GeomGetResultWKTClob(GEO_HANDLE      geoHandle,
                        LOB_RESULT_LOCATOR* geoClob)

```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### LOB\_RESULT\_LOCATOR *geoClob*

Pointer to the LOB locator that is used to set the ST\_Geometry type return value.

### Usage Notes

The FNC\_GeomGetResultWKTClob function is used get a result LOB locator that is used to set the ST\_Geometry return value. The ST\_Geometry Handle *geoHandle* is passed as input, and a LOB locator



*geoClob* is returned from the function. The user can then use LOB FNC routines to append to the CLOB value using the *geoClob* locator.

---

**Note:**

Use the Latin character set to append values to the CLOB. The Well-Known Text value should be written to the CLOB.

---

The `FNC_GeomGetResultWKTClob` function should be called only for LOB-based `ST_Geometry` values. If used with inline `ST_Geometry` values, a 7579 error is generated: `FNC_GeomGetResultWKTClob` can only be called on LOB-based `ST_Geometry` type.

**Example: Using `FNC_GeomGetResultWKTClob` and `FNC_GeomSetWKTClob` to retrieve and set the value of a return `ST_Geometry`**

The following function takes an `ST_Geometry` parameter and returns an `ST_Geometry` that has the same value. It first retrieves the Well-Known Text format from the input geometry. It then retrieves the LOB result locator from the return `ST_Geometry`, appends the WKT to the CLOB, and then sets the CLOB into the return `ST_Geometry`.

```

REPLACE FUNCTION geo_wkt(p1 ST_Geometry)
RETURNS ST_Geometry
SPECIFIC geo_wkt
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geo_wkt!geo_wkt.c!F!geo_wkt';

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void geo_wkt( GEO_HANDLE *geo_in,
              GEO_HANDLE *geo_out,
              int         *indicator_geo_in,
              int         *indicator_geo_out,
              char         sqlstate[6],
              SQL_TEXT     extname[129],
              SQL_TEXT     specific_name[129],
              SQL_TEXT     error_message[257])
{
    LOB_RESULT_LOCATOR geoClob_out;
    LOB_LOCATOR geoClob_in;
    BYTE buffer[64000];
    LOB_CONTEXT_ID id;

```

```

FNC_LobLength_t actlen;
int trunc_err = 0;
FNC_LobLength_t clob_length;
int srid = -1;
int max_length, max_length2, numLobs, numLobs2;

/* If input is null set return value to null. */
if (*indicator_geo_in == -1 )
{
    *indicator_geo_out = -1;
    return;
}
/* Get the Input Geometry Info */
FNC_GetGeometryInfo(*geo_in,&max_length, &numLobs);

/* Get the Output Geometry Info */
FNC_GetGeometryInfo(*geo_out,&max_length2, &numLobs2);

/*
 * Copy geo_in ST_Geometry value to geo_out ST_Geometry value.
 */

if(numLobs ==1 && numLobs2==1)
{
    /* Get the LOB_RESULT_LOCATOR of geo_out. */
    FNC_GeomGetResultWKTClob(*geo_out, &geoClob_out);

    /* Get the LOB_LOCATOR and srid of the input ST_Geometry value. */
    FNC_GeomGetWKTClob(*geo_in, &geoClob_in, &srid);

    FNC_LobOpen(geoClob_in, &id, 0, 0);

    clob_length = FNC_GetLobLength(geoClob_in);

    while (FNC_LobRead(id, buffer, clob_length, &actlen) == 0 && !trunc_err)
        trunc_err = FNC_LobAppend(geoClob_out, buffer, actlen, &actlen);

    FNC_LobClose(id);

    /* Set the return ST_Geometry value */
    FNC_GeomSetWKTClob(*geo_out, geoClob_out, srid);

```

```

    }
}

```

## FNC\_GeomGetWKB

Returns the Well-Known Binary (WKB) representation of the Geometry as a binary string.

### Syntax

```

void
FNC_GeomGetWKB(GEO_HANDLE  geoHandle,
               byte *       wkbBuffer,
               FNC_GeomSize_t wkbBufferSize,
               FNC_GeomSize_t *wkbSize,
               int           srid)

```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input parameter to an external routine.

#### *wkbBuffer*

A pointer to the buffer that holds the WKB value.

#### *wkbBufferSize*

Size in bytes of the *wkbBuffer* passed to the FNC routine.

#### *wkbSize*

Size in bytes of the WKB data returned by the routine.

#### *srid*

Spatial Reference System Identifier of the Geometry returned by the routine.

### Usage Notes

FNC\_GeomGetWKB can be used to get the Well-Known Binary representation of the Geometry. The GEO\_HANDLE for the Geometry is passed in as input along with a pointer to a buffer (*wkbBuffer*) and the buffer size (*wkbBufferSize*). The *wkbBuffer* size must be large enough to hold the WKB data. To allocate the appropriate buffer size, the FNC\_GetGeometryInfo function or FNC\_GetWKBSize function can be used to find the Geometry length.

The routine fills up the *wkbBuffer* with the WKB representation of the geometry. The *wkbSize* and *srid* fields are also filled in by the FNC routine.

This routine should be called only on inline ST\_Geometry values. If invoked on a LOB-based ST\_Geometry value, a 7579 error is generated: FNC\_GeomGetWKB is only valid for inline ST\_Geometry values.

### Example: Using FNC\_GeomGetWKB to retrieve a ST\_Geometry WKB

The following example function takes a ST\_Geometry parameter and retrieves the Well-Known Binary format for the geometry. It returns the WKB retrieved as a BLOB.

```
CREATE FUNCTION geomUDF2(P1 st_geometry(1000))
RETURNS BLOB AS LOCATOR
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF2!geomUDF2.c';

void geomUDF2(GEO_HANDLE *geom_handle,
              LOB_RESULT_LOCATOR *return_handle,
              int*          indicator_thisGeom,
              int*          indicator_returnLOB,
              char          sqlstate[6],
              SQL_TEXT      extname[129],
              SQL_TEXT      specific_name[129],
              SQL_TEXT      error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t wkbSize;
    byte* wkbBuffer;
    FNC_GeomSize_t wkbBufferSize;
    FNC_LobLength_t actlen;
    int srid;

    /* Get the Geometry WKT Size */
    geomsz = FNC_GeomGetWKBSz(*geom_handle);

    wkbBuffer = (byte*)FNC_Malloc(geomsz);
    wkbBufferSize = geomsz;
    /* Get the WKB value */
    FNC_GeomGetWKB(*geom_handle,wkbBuffer, wkbBufferSize,
                  &wkbSize, &srid);
}
```

```

/* Append the WKB to a LOB */
FNC_LobAppend(*return_handle, wkbBuffer, wkbBufferSize,
              &actlen);
*indicator_returnLob = 0;
FNC_free(wkbBuffer);
}

```

## FNC\_GeomGetWKBBlob

Returns the Well-Known Binary (WKB) representation of the Geometry as a BLOB.

### Syntax

```

void
FNC_GeomGetWKBBlob(GEO_HANDLE  geoHandle,
                   LOB_LOCATOR *geoBlob,
                   int  *srid)

```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input parameter to an external routine.

#### *geoBlob*

LOB locator for the BLOB representation of the ST\_Geometry type.

#### *srid*

Spatial Reference System Identifier of the Geometry.

### Usage Notes

FNC\_GeomGetWKBBlob gets a LOB locator for the Well-Known Binary BLOB representation of the ST\_Geometry type. The ST\_Geometry Handle *geoHandle* is passed as input. The LOB locator value *geoBlob* is returned from the function. The user can then use LOB FNC routines to read from the *geoBlob* locator.

The FNC\_GeomGetWKBBlob function can only be used with LOB-based ST\_Geometry values. If used with an inline ST\_Geometry value, a 7579 error is generated: FNC\_GeomGetWKBBlob is only valid for LOB-based ST\_Geometry values. Use FNC\_GeomGetWKB() instead.

### Example: Using FNC\_GeomGetWKBBlob to retrieve a ST\_Geometry WKB

The following example function takes an ST\_Geometry parameter and retrieves the Well-Known Binary format for the geometry. It returns the WKB retrieved as a BLOB.

```

REPLACE FUNCTION getwkb_udf(p1 ST_Geometry)
RETURNS BLOB AS LOCATOR
SPECIFIC getwkb_udf
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getwkb_udf!getwkb.c!F!getwkb_udf';

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void getwkb_udf(GEO_HANDLE *geo_in,
                LOB_RESULT_LOCATOR *outBlob,
                int *indicator_geo_in,
                int *indicator_outBlob,
                char sqlstate[6],
                SQL_TEXT extname[129],
                SQL_TEXT specific_name[129],
                SQL_TEXT error_message[257])
{
    LOB_LOCATOR geoBlob_in;
    BYTE buffer[64000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_LobLength_t blob_length;
    int srid = -1;
    int max_length, numLobs;

    /* If input is null set return value to null. */
    if (*indicator_geo_in == -1 )
    {
        *indicator_outBlob = -1;
        return;
    }
    /* Get the Geometry Info */
    FNC_GetGeometryInfo(*geo_in,&max_length, &numLobs);

    /*
     * Copy geo_in ST_Geometry value to outClob value.
     */

```

```

    */
    if(numLobs == 1)
    {

        /*
        * Copy geo_in ST_Geometry value to outBlob value.
        *
        */

        /* Get the LOB_LOCATOR and srid of the input ST_Geometry value. */
        FNC_GeomGetWKBBlob(*geo_in, &geoBlob_in, &srid);

        FNC_LobOpen(geoBlob_in, &id, 0, 0);
        blob_length = FNC_GetLobLength(geoBlob_in);
        while (FNC_LobRead(id, buffer, blob_length, &actlen) == 0 && !trunc_err)
            trunc_err = FNC_LobAppend(*outBlob, buffer, actlen, &actlen);
        FNC_LobClose(id);
    }
}

```

## FNC\_GeomGetWKBSize

Returns the length in bytes of the Well-Known Binary (WKB) representation of the Geometry.

### Syntax

```

FNC_GeomSize_t
FNC_GeomGetWKBSize(GEO_HANDLE  geoHandle)

```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input or output parameter to an external routine.

### Usage Notes

FNC\_GeomGetWKBSize gets the size of the WKB representation of the Geometry. The GEO\_HANDLE for the Geometry is passed in as input and the size of the WKB representation of the Geometry is returned to the caller.

#### **Example: Using FNC\_GeomGetWKB to retrieve a ST\_Geometry WKB**

The following example function takes a ST\_Geometry parameter and retrieves the Well-Known Binary format for the geometry. It returns the WKB retrieved as a BLOB.

```

CREATE FUNCTION geomUDF2(P1 st_geometry(1000))
RETURNS BLOB AS LOCATOR
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF2!geomUDF2.c';

void geomUDF2(GEO_HANDLE *geom_handle,
              LOB_RESULT_LOCATOR *return_handle,
              int*          indicator_thisGeom,
              int*          indicator_returnLOB,
              char          sqlstate[6],
              SQL_TEXT      extname[129],
              SQL_TEXT      specific_name[129],
              SQL_TEXT      error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t wkbSize;
    byte* wkbBuffer;
    FNC_GeomSize_t wkbBufferSize;
    FNC_LobLength_t actlen;
    int srid;

    /* Get the Geometry WKT Size */
    geomsz = FNC_GeomGetWKBSz(*geom_handle);

    wkbBuffer = (byte*)FNC_Malloc(geomsz);
    wkbBufferSize = geomsz;
    /* Get the WKB value */
    FNC_GeomGetWKB(*geom_handle,wkbBuffer, wkbBufferSize,
                  &wkbSize, &srid);

    /* Append the WKB to a LOB */
    FNC_LobAppend(*return_handle, wkbBuffer, wkbBufferSize,
                  &actlen);
    *indicator_returnLob = 0;
    FNC_free(wkbBuffer);
}

```



## FNC\_GeomGetWKT

Returns the Well-Known Text (WKT) representation of the Geometry as a null terminated character string in the Latin character set.

### Syntax

```
void
FNC_GeomGetWKT(GEO_HANDLE  geoHandle,
               unsigned char *wktBuffer,
               FNC_GeomSize_t wktBufferSize,
               FNC_GeomSize_t *wktSize,
               int *srid)
```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input parameter to an external routine.

#### *wktBuffer*

A pointer to the buffer that will hold the WKT string.

#### *wktBufferSize*

Size in bytes of the *wktBuffer* passed to the FNC routine.

#### *wktSize*

Size in bytes of the WKT string returned by the routine.

#### *srid*

Spatial Reference System Identifier of the Geometry returned by the routine.

### Usage Notes

FNC\_GeomGetWKT gets the Well Known Text representation of the Geometry. The GEO\_HANDLE for the Geometry is passed in as input along with a pointer to a buffer (*wktBuffer*) and the buffer size (*wktBufferSize*). The *wktBuffer* size must be large enough to hold the WKT value. To determine the appropriate buffer size, use the FNC\_GetGeometryInfo function or FNC\_GetWKTSize function call to get the Geometry length.

The routine fills up the *wktBuffer* with the WKT representation. The *wktSize* and *srid* fields are also filled in by the FNC routine.

This FNC function should be called only on inline ST\_Geometry values. If invoked for a LOB-based ST\_Geometry value, a 7579 error is generated: FNC\_GeomGetWKT is only valid for inline ST\_Geometry values.

### Example: Using FNC\_GeomGetWKT to retrieve a ST\_Geometry WKT

The following example function takes a ST\_Geometry parameter and retrieves the Well Known Text format for the geometry. It returns the string back in the ST\_Geometry return parameter.

```
CREATE FUNCTION geomUDF1(P1 st_geometry(1000))
RETURNS ST_GEOMETRY(1000)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF1!geomUDF1.c';

void geomUDF1(GEO_HANDLE* geom_handle,
              GEO_HANDLE* return_handle,
              int* indicator_thisGeom,
              int* indicator_returnGeom,
              char sqlstate[6],
              SQL_TEXT extname[129],
              SQL_TEXT specific_name[129],
              SQL_TEXT error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t wktsz;
    char* wktBuffer;
    FNC_GeomSize_t wktBufferSize;
    int srid , max_length, numLobs;

    /* Get the Geometry Info */
    FNC_GetGeometryInfo(*geom_handle,&max_length, &numLobs);
    if(numLobs == 0)
    {
        /* Get the Geometry WKT Size */
        geomsz = FNC_GeomGetWKTSz(*geom_handle);

        /* Allocate the buffer to hold the WKT. Note that we add
           1 for the null termination character. */
        wktBuffer = (char*)FNC_Malloc(geomsz + 1);
        wktBufferSize = geomsz + 1;
    }
}
```

```

/* Get the WKT string */
FNC_GeomGetWKT(*geom_handle, wktBuffer, wktBufferSize,
               &wktSize, &srid);

/* Set the WKT string as return value */
FNC_GeomSetWKT(*return_handle, wktBuffer, wktSize, 0);
*indicator_returnGeom = 0;
FNC_free(wkbBuffer);
}
}

```

## FNC\_GeomGetWKTCllob

Returns the Well-Known Text (WKT) representation of the Geometry as a CLOB in the Latin character set.

### Syntax

```

void
FNC_GeomGetWKTCllob(GEO_HANDLE  geoHandle,
                    LOB_LOCATOR *geoClob,
                    int  *srid)

```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input parameter to an external routine.

#### *geoClob*

LOB locator for the CLOB representation of the ST\_Geometry type.

#### *srid*

Spatial Reference System Identifier of the Geometry.

### Usage Notes

FNC\_GeomGetWKTCllob is used to get a LOB locator for the CLOB representation of the ST\_Geometry type. The ST\_Geometry Handle *geoHandle* is passed as input. The LOB locator value *geoClob* is returned from the function. The user can then use LOB FNC routines to read from the locator.

The FNC\_GetGeometryInfo function can be used to determine if the ST\_Geometry instance is a LOB-based value. If the instance is not a LOB (i.e. *numLobs* = 0), then use the non-LOB FNC\_GeomGetWKT function.

If the `FNC_GeomGetWKTCLob` function is used on an inline `ST_Geometry` value, a 7579 error is generated: `FNC_GeomGetWKTCLob` is only valid for LOB-based `ST_Geometry` values. Use `FNC_GeomSetWKT()` instead.

### Example: Using `FNC_GeomGetWKTCLob` to retrieve a `ST_Geometry` WKT

The following function takes an `ST_Geometry` parameter and retrieves the Well-Known Text format for the geometry. It returns the string back as a CLOB.

```

REPLACE FUNCTION getwkt_udf(p1 ST_Geometry)
RETURNS CLOB AS LOCATOR
SPECIFIC getwkt_udf
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getwkt_udf!getwkt.c!F!getwkt_udf';

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void getwkt_udf(GEO_HANDLE *geo_in,
                LOB_RESULT_LOCATOR *outClob,
                int *indicator_geo_in,
                int *indicator_outClob,
                char sqlstate[6],
                SQL_TEXT extname[129],
                SQL_TEXT specific_name[129],
                SQL_TEXT error_message[257])
{
    LOB_LOCATOR geoClob_in;
    BYTE buffer[64000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_LobLength_t clob_length;
    int srid = -1;
    int max_length;
    int numLobs;

    /* If input is null set return value to null. */
    if (*indicator_geo_in == -1 )
    {

```

```

        *indicator_outClob = -1;
        return;
    }
    /* Get the Geometry Info */
    FNC_GetGeometryInfo(*geo_in,&max_length, &numLobs);

    /*
     * Copy geo_in ST_Geometry value to outClob value.
     */
    /*
    if(numLobs == 1)
    {
        /* Get the LOB_LOCATOR and srid of the input ST_Geometry value. */
        FNC_GeomGetWKTClob(*geo_in, &geoClob_in, &srid);

        FNC_LobOpen(geoClob_in, &id, 0, 0);
        clob_length = FNC_GetLobLength(geoClob_in);
        while (FNC_LobRead(id, buffer, clob_length, &actlen) == 0
                && !trunc_err)
            trunc_err = FNC_LobAppend(*outClob, buffer, actlen, &actlen);
        FNC_LobClose(id);
    }
}

```

## FNC\_GeomGetWKTSIZE

Returns the length in bytes of the Well-Known Text (WKT) representation of the Geometry.

### Syntax

```
FNC_GeomSize_t  FNC_GeomGetWKTSIZE(GEO_HANDLE  geoHandle)
```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input or output parameter to an external routine.

### Usage Notes

FNC\_GeomGetWKTSIZE gets the size of the WKT representation of the Geometry. The GEO\_HANDLE for the Geometry is passed in as input and the size of the WKT representation of the Geometry is returned to the caller.

**Example: Using FNC\_GeomGetWKT to retrieve a ST\_Geometry WKT**

The following example function takes a ST\_Geometry parameter and retrieves the Well Known Text format for the geometry. It returns the string back in the ST\_Geometry return parameter.

```
CREATE FUNCTION geomUDF1(P1 st_geometry(1000))
RETURNS ST_GEOMETRY(1000)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF1!geomUDF1.c';

void geomUDF1(GEO_HANDLE* geom_handle,
              GEO_HANDLE* return_handle,
              int* indicator_thisGeom,
              int* indicator_returnGeom,
              char sqlstate[6],
              SQL_TEXT extname[129],
              SQL_TEXT specific_name[129],
              SQL_TEXT error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t wktsz;
    char* wktBuffer;
    FNC_GeomSize_t wktBufferSize;
    int srid , max_length, numLobs;

    /* Get the Geometry Info */
    FNC_GetGeometryInfo(*geom_handle,&max_length, &numLobs);
    if(numLobs == 0)
    {
        /* Get the Geometry WKT Size */
        geomsz = FNC_GeomGetWKTSz(*geom_handle);

        /* Allocate the buffer to hold the WKT. Note that we add
           1 for the null termination character. */
        wktBuffer = (char*)FNC_Malloc(geomsz + 1);
        wktBufferSize = geomsz + 1;

        /* Get the WKT string */
        FNC_GeomGetWKT(*geom_handle,wktBuffer, wktBufferSize,
                      &wktsz, &srid);
    }
}
```

```

    /* Set the WKT string as return value */
    FNC_GeomSetWKT(*return_handle, wktBuffer, wktSize,0);
    *indicator_returnGeom = 0;
    FNC_free(wkbBuffer);
  }
}

```

## FNC\_GeomSetWKB

Set the value of a Geometry type using a Well-Known Binary (WKB) representation of the Geometry.

### Syntax

```

void
FNC_GeomSetWKB(GEO_HANDLE      geoHandle,
                byte*           wkb,
                FNC_GeomSize_t  wkbSize,
                int              srid)

```

### Syntax Elements

#### *geoHandle*

A handle to a Geometry type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *wkb*

A pointer to the WKB value to be set.

#### *wkbSize*

Size in bytes of the WKB value.

#### *srid*

Spatial Reference System Identifier of the Geometry to be set.

### Usage Notes

FNC\_GeomSetWKB sets the value of a Geometry using a WKB byte string. The *wkb* parameter containing the WKB string, the *wkbSize* and the SRID to be set for the Geometry are passed as input.

FNC\_GeomSetWKB should be called only for inline ST\_Geometry values. If used with a LOB-based ST\_Geometry value, a 7579 error is generated: FNC\_GeomSetWKB is only valid for inline ST\_Geometry values.

**Example: Using FNC\_GeomGetWKB and FNC\_GeomSetWKB to get and set a geometry WKB**

The following UDF takes a ST\_Geometry parameter as input, retrieves the WKB representation, and sets the return ST\_Geometry value with it. It assumes the Geometry size is small enough to be read using a single read.

```
CREATE FUNCTION geomUDF3(P1 st_geometry)
RETURNS ST_GEOMETRY
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF3!geomUDF3.c';

void geomUDF3(GEO_HANDLE* geom_handle,
              GEO_HANDLE* return_handle,
              int* indicator_thisGeom,
              int* indicator_returnValue,
              char  sqlstate[6],
              SQL_TEXT      extname[129],
              SQL_TEXT      specific_name[129],
              SQL_TEXT      error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t bytesRead;
    FNC_GeomSize_t offset;
    FNC_GeomSize_t size;
    FNC_GeomSize_t wkbSize;
    byte* wkbBuffer;
    FNC_GeomSize_t wkbBufferSize;
    int srid;

    /* Get the Geometry WKT Size */
    geomsz = FNC_GeomGetWKBSize(*geom_handle);

    /* Read the WKB value */
    wkbBuffer = (byte*)FNC_Malloc(geomsz);
    wkbBufferSize = geomsz;
    size = geomsz;
    offset = 0;
    /* Get the WKB value */
    FNC_GeomGetWKB(*geom_handle, wkbBuffer, wkbBufferSize,
    &wkbSize, &srid);
}
```



```

    /* Set the return value */
    FNC_GeomSetWKB(*return_handle, wkbBuffer, wkbSize,0);
    *indicator_returnValue = 0;
    FNC_free(wkbBuffer);
}

```

## FNC\_GeomSetWKBBlob

Set the value of a Geometry type using a Well-Known Binary (WKB) BLOB represented by a LOB\_RESULT\_LOCATOR.

### Syntax

```

void
FNC_GeomGetWKBBlob(GEO_HANDLE      geoHandle,
                   LOB_RESULT_LOCATOR geoBlob,
                   int               srid)

```

### Syntax Elements

#### *geoHandle*

A handle to a Geometry type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *geoBlob*

LOB locator for the WKB BLOB containing the return value for the ST\_Geometry type.

#### *srid*

Spatial Reference System Identifier of the Geometry to be set.

### Usage Notes

FNC\_GeomSetWKBBlob sets the ST\_Geometry return/OUT/INOUT parameter value using a LOB result locator. The ST\_Geometry Handle *geoHandle* is passed as input along with a LOB result locator *geoBlob*. The *geoBlob* locator value is obtained by calling FNC\_GeomGetResultWKBBlob, and LOB FNC routines are used to append the WKB to the BLOB. It is then passed to FNC\_GeomSetWKBBlob.

#### Note:

The Blob returned by this routine must contain well-formed WKB data else the FNC routine will return an error ERRAMPFNCUDTBADARG.

FNC\_GeomSetWKBBlob should be used only with a LOB-based ST\_Geometry value. Otherwise a 7579 error is generated: FNC\_GeomSetWKBBlob is only valid for LOB-based ST\_Geometry values. Use FNC\_GeomSetWKB() instead.

**Example: Using FNC\_GeomGetResultWKBBlob and FNC\_GeomSetWKBBlob to retrieve and set the value of a return ST\_Geometry**

The following function takes a ST\_Geometry parameter and returns an ST\_Geometry that has the same value. It first retrieves the Well-Known Binary format from the input geometry. It then retrieves the LOB result locator from the return ST\_Geometry, appends the WKB to it, and then sets the value of the return ST\_Geometry.

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void geo_wkb( GEO_HANDLE *geo_in,
              GEO_HANDLE *geo_out,
              int *indicator_geo_in,
              int *indicator_geo_out,
              char sqlstate[6],
              SQL_TEXT extname[129],
              SQL_TEXT specific_name[129],
              SQL_TEXT error_message[257])
{
    LOB_RESULT_LOCATOR geoBlob_out;
    LOB_LOCATOR geoBlob_in;
    BYTE buffer[64000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_LobLength_t blob_length;
    int srid = -1;
    int max_length, max_length2, numLobs, numLobs2;

    /* If input is null set return value to null. */
    if (*indicator_geo_in == -1 )
    {
        *indicator_geo_out = -1;
        return;
    }

    /* Get the Input Geometry Info */
    FNC_GetGeometryInfo(*geo_in,&max_length, &numLobs);
```

```

/* Get the Output Geometry Info */
FNC_GetGeometryInfo(*geo_out,&max_length2, &numLobs2);

/*
 * Copy geo_in ST_Geometry value to geo_out ST_Geometry value.
 */
if(numLobs ==1 && numLobs2 ==1)
{
    /* Get the LOB_RESULT_LOCATOR of geo_out. */
    FNC_GeomGetResultWKBBlob(*geo_out, &geoBlob_out);

    /* Get the LOB_LOCATOR and srid of the input ST_Geometry value. */
    FNC_GeomGetWKBBlob(*geo_in, &geoBlob_in, &srid);

    FNC_LobOpen(geoBlob_in, &id, 0, 0);
    blob_length = FNC_GetLobLength(geoBlob_in);
    while (FNC_LobRead(id, buffer, blob_length, &actlen) == 0
           && !trunc_err)
        trunc_err = FNC_LobAppend(geoBlob_out, buffer, actlen, &actlen);
    FNC_LobClose(id);

    /* Set the return ST_Geometry value */
    FNC_GeomSetWKBBlob(*geo_out, geoBlob_out, srid);
}
}

```

## FNC\_GeomSetWKT

Set the value of a Geometry type using a Well-Known Text (WKT) string representation of the Geometry in the Latin character set.

### Syntax

```

void
FNC_GeomSetWKT(GEO_HANDLE      geoHandle,
               unsigned char*   wkt,
               FNC_GeomSize_t   wktSize,
               int               srid)

```

## Syntax Elements

### *geoHandle*

A handle to a Geometry type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

### *wkt*

A pointer to the WKT string in the Latin character set.

### *wktSize*

Size in bytes of the WKT string.

### *srid*

Spatial Reference System Identifier of the Geometry to be set.

## Usage Notes

The FNC\_GeomSetWKT function sets the value of a Geometry using a WKT string representation. The *wkt* parameter containing the WKT string, the *wktSize*, and the SRID to be set for the Geometry are passed as input.

The FNC\_GeomSetWKT function should be called only for inline ST\_Geometry values. If used with a LOB-based ST\_Geometry value, a 7579 error is generated: FNC\_GeomSetWKT is only valid for inline ST\_Geometry values.

### Example: Using FNC\_GeomGetWKT to retrieve a ST\_Geometry WKT

The following example function takes a ST\_Geometry parameter and retrieves the Well Known Text format for the geometry. It returns the string back in the ST\_Geometry return parameter.

```
CREATE FUNCTION geomUDF1(P1 st_geometry(1000))
RETURNS ST_GEOMETRY(1000)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF1!geomUDF1.c';

void geomUDF1(GEO_HANDLE* geom_handle,
              GEO_HANDLE* return_handle,
              int* indicator_thisGeom,
              int* indicator_returnGeom,
              char sqlstate[6],
              SQL_TEXT extname[129],
```

```

        SQL_TEXT    specific_name[129],
        SQL_TEXT    error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t wktSz;
    char* wktBuffer;
    FNC_GeomSize_t wktBufferSize;
    int srid , max_length, numLobs;

    /* Get the Geometry Info */
    FNC_GetGeometryInfo(*geom_handle,&max_length, &numLobs);
    if(numLobs == 0)
    {
        /* Get the Geometry WKT Size */
        geomsz = FNC_GeomGetWKTSz(*geom_handle);

        /* Allocate the buffer to hold the WKT. Note that we add
           1 for the null termination character. */
        wktBuffer = (char*)FNC_Malloc(geomsz + 1);
        wktBufferSize = geomsz + 1;

        /* Get the WKT string */
        FNC_GeomGetWKT(*geom_handle,wktBuffer, wktBufferSize,
                      &wktSz, &srid);

        /* Set the WKT string as return value */
        FNC_GeomSetWKT(*return_handle, wktBuffer, wktSz,0);
        *indicator_returnGeom = 0;
        FNC_free(wkbBuffer);
    }
}

```

## FNC\_GeomSetWKTCLob

Set the value of a Geometry type using a Well-Known Text (WKT) CLOB represented by a LOB\_RESULT\_LOCATOR.

### Syntax

```

void
FNC_GeomSetWKTCLob(GEO_HANDLE      geoHandle,
```

```
LOB_RESULT_LOCATOR  geoClob,
int                  srid)
```

## Syntax Elements

### *geoHandle*

A handle to a Geometry type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

### *geoClob*

LOB locator for the WKT CLOB containing the return value for the ST\_Geometry type in the Latin character set.

### *srid*

Spatial Reference System Identifier of the Geometry to be set.

## Usage Notes

The FNC\_GeomSetWKTclob function sets the ST\_Geometry return/OUT/INOUT parameter value using a LOB result locator. The ST\_Geometry Handle *geoHandle* is passed as input along with a LOB result locator *geoClob*.. The *geoClob* locator value is obtained by calling FNC\_GeomGetResultWKTclob, and LOB FNC routines are used to append the WKT value to the CLOB. It is then passed to the FNC\_GeomSetWKTclob routine. The CLOB should be in the Latin character set.

### Note:

The CLOB returned by this routine must contain well-formed WKT data else the FNC routine will return an error ERRAMPFNCDTBADARG.

FNC\_GeomSetWKTclob should be used only with a LOB-based ST\_Geometry value. Otherwise a 7579 error is generated: FNC\_GeomSetWKTclob is only valid for LOB-based ST\_Geometry values. Use FNC\_GeomSetWKT() instead.

### **Example: Using FNC\_GeomGetResultWKTclob and FNC\_GeomSetWKTclob to retrieve and set the value of a return ST\_Geometry**

The following function takes an ST\_Geometry parameter and returns an ST\_Geometry that has the same value. It first retrieves the Well-Known Text format from the input geometry. It then retrieves the LOB result locator from the return ST\_Geometry, appends the WKT to the CLOB, and then sets the CLOB into the return ST\_Geometry.

```
REPLACE FUNCTION geo_wkt(p1 ST_Geometry)
RETURNS ST_Geometry
SPECIFIC geo_wkt
NO SQL
```

```

PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geo_wkt!geo_wkt.c!F!geo_wkt';

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void geo_wkt( GEO_HANDLE *geo_in,
              GEO_HANDLE *geo_out,
              int         *indicator_geo_in,
              int         *indicator_geo_out,
              char         sqlstate[6],
              SQL_TEXT     extname[129],
              SQL_TEXT     specific_name[129],
              SQL_TEXT     error_message[257])
{
    LOB_RESULT_LOCATOR geoClob_out;
    LOB_LOCATOR geoClob_in;
    BYTE buffer[64000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_LobLength_t clob_length;
    int srid = -1;
    int max_length, max_length2, numLobs, numLobs2;

    /* If input is null set return value to null. */
    if (*indicator_geo_in == -1 )
    {
        *indicator_geo_out = -1;
        return;
    }
    /* Get the Input Geometry Info */
    FNC_GetGeometryInfo(*geo_in,&max_length, &numLobs);

    /* Get the Output Geometry Info */
    FNC_GetGeometryInfo(*geo_out,&max_length2, &numLobs2);

    /*
     * Copy geo_in ST_Geometry value to geo_out ST_Geometry value.
     */
}

```

```

if(numLobs ==1 && numLobs2==1)
{
    /* Get the LOB_RESULT_LOCATOR of geo_out. */
    FNC_GeomGetResultWKTClob(*geo_out, &geoClob_out);

    /* Get the LOB_LOCATOR and srid of the input ST_Geometry value. */
    FNC_GeomGetWKTClob(*geo_in, &geoClob_in, &srid);

    FNC_LobOpen(geoClob_in, &id, 0, 0);

    clob_length = FNC_GetLobLength(geoClob_in);

    while (FNC_LobRead(id, buffer, clob_length, &actlen) == 0 && !trunc_err)
        trunc_err = FNC_LobAppend(geoClob_out, buffer, actlen, &actlen);

    FNC_LobClose(id);

    /* Set the return ST_Geometry value */
    FNC_GeomSetWKTClob(*geo_out, geoClob_out, srid);
}
}

```

## FNC\_GetAmphHash

Returns values that hash to the specified AMPs.

### Syntax

```

void
FNC_GetAmphHash (int **amphash,
                 int    size)

```

### Syntax Elements

#### *amphash*

IN/OUT parameter

*amphash*[*n*][0] is the AMP number.

*amphash*[*n*][1] will be returned with the value that hashes to the AMP.

#### *size*

IN parameter



The size ( $n$ ) of the amphash array.

## Usage Notes

This routine is callable on an AMP or PE vproc.

## FNC\_GetAnyTypeParamInfo [Deprecated]

This function is deprecated because it truncates object names to 30 characters. However, it remains available to support legacy applications. For current and future development, use the corresponding function that includes "EON" in the function name. For example, use FNC\_DbsInfo\_EON instead of FNC\_DbsInfo.

Returns information about the TD\_ANYTYPE input and output arguments passed to an external routine.

## Syntax

```
void
FNC_GetAnyTypeParamInfo(int          bufsize,
                        int          *numAnyTypeParams,
                        anytype_param_info_t *AnyTypeAttributeArray);
```

### AnyTypeAttributeArray

Defined in sqltypes\_td.h as:

```
typedef struct anytype_param_info_t {
    INTEGER_td      paramIndex;
    dtype_et        datatype;
    dmode_et        direction;
    INTEGER_td      max_length;
    FNC_LobLength_t lob_length;
    SMALLINT        total_interval_digits;
    SMALLINT        num_fractional_digits;
    charset_et      charset;
    [ CHARACTER     UDTName[FNC_MAXNAMELEN]; ]
    SMALLINT        udt_indicator;
} anytype_param_info_t;
```

## Syntax Elements

### *bufsize*

Size of the *AnyTypeAttributeArray* buffer.

***numAnyTypeParams***

Number of input and output arguments of TD\_ANYTYPE data type.

***AnyTypeAttributeArray***

Pointer to the buffer that will hold the information about the TD\_ANYTYPE parameters.

For more information about dtype\_et, total\_interval\_digits, num\_fractional\_digits, or charset\_et, see [FNC\\_GetStructuredAttributeInfo \[Deprecated\]](#).

For more information about dmode\_et, see [FNC\\_CallSP](#).

***paramIndex***

the parameter positional index starting from 1. For the return parameter, the index will be -1.

***datatype***

the data type of the parameter.

***direction***

indicates whether the parameter is an input, output, or an INOUT parameter.

***max\_length***

the maximum length in bytes that this parameter may utilize. If the parameter is a LOB type, *max\_length* indicates the LOB\_REF length. *lob\_length* provides the length of the LOB data itself.

***lob\_length***

the maximum length of the parameter if it is a LOB type parameter.

***total\_interval\_digits***

the precision value for certain data types. For example, the value *n* in a DECIMAL(*n,m*) type or in INTERVAL DAY(*n*) TO SECOND(*m*). The list of types that use this value is the same as that of attribute\_info\_t.total\_interval\_digits.

***num\_fractional\_digits***

the precision or scale value for certain data types. For example, the value *m* in a DECIMAL(*n,m*) type or in INTERVAL DAY(*n*) TO SECOND(*m*). The list of types that use this value is the same as that of attribute\_info\_t.num\_fractional\_digits.

***charset***

the server character set associated with a character data type.

**UDTName**

[Required if parameter is a UDT, disallowed otherwise]. The name of the UDT.

**udt\_indicator**

Indicates kind of UDT:

<i>udt_indicator</i>	UDT Kind
0	Not a UDT
1	Structured UDT
2	Distinct UDT For distinct types, the data type returns the underlying type of the distinct type UDT.
3	Internal UDT
4	Array Type

## Usage Notes

You can use `FNC_GetAnyTypeParamInfo` but it truncates a retrieved UDT Name after 30 characters. Use `FNC_GetAnyTypeParamInfo_eon` instead of `FNC_GetAnyTypeParamInfo` for longer object names.

To get information about the `TD_ANYTYPE` input and output arguments passed to an external routine:

1. Declare an integer variable, such as *numAnytypeParams*.
2. Allocate sufficient memory for the *AnyTypeAttributeArray* buffer to hold information for all of the `TD_ANYTYPE` input and result parameters declared in your routine. This should be an array of `anytype_param_info_t` structures, where each element of the array will contain the information for a particular `TD_ANYTYPE` parameter.
3. Call `FNC_GetAnyTypeParamInfo()` and pass in a pointer to the *AnyTypeAttributeArray* buffer along with the size of the buffer, and a pointer to *numAnytypeParams*.

The `FNC_GetAnyTypeParamInfo` function fills the *AnyTypeAttributeArray* buffer with information about each `TD_ANYTYPE` parameter. The information for the return type of a `TD_ANYTYPE` result parameter is contained in the last element of the array. The number of `TD_ANYTYPE` parameters is returned in the *numAnyTypeParams* parameter.

---

**Note:**

Be sure to free the memory allocated to the *AnyTypeAttributeArray* buffer when you have completed processing the information.

---

## Example

The following code excerpt is an example of a UDF that calls the `FNC_GetAnyTypeParamInfo` function to retrieve information about the `TD_ANYTYPE` arguments.

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
void ScalarFncWAnytype(void *input1,
                      ...,
                      void *result,
                      char sqlstate[6])
{
    anytype_param_info_t paraminfo[2];
    FNC_GetAnyTypeParamInfo(2*sizeof(anytype_param_info_t), &numAnyType, paraminfo);
    if(((paraminfo[1].datatype == CHAR_DT) ||
        (paraminfo[1].datatype == VARCHAR_DT)) &&
        (paraminfo[1].paramIndex == -1) &&
        (paraminfo[1].direction == OUT_PM) &&
        (paraminfo[1].charset == LATIN_CT))
    {
        /* Process the value based upon the data type */
        switch(paraminfo[0].datatype)
        {
            case BYTEINT_DT:
                strcpy ((char *)result, "Byteint");
                break;
                /* Do some processing */
                ...
        }
    }
}
```

The following shows the corresponding SQL function definition:

```
REPLACE FUNCTION ScalarFncWAnytype ( p1 TD_ANYTYPE, ...)
  RETURNS TD_ANYTYPE
  NO SQL
  PARAMETER STYLE SQL
  RETURNS NULL ON NULL INPUT
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!ScalarFncWAnytype!udfsrc/ScalarFncWAnytype.c';
```

For additional examples of functions that call `FNC_GetAnyTypeParamInfo()` to retrieve `TD_ANYTYPE` argument information, see [C Scalar Function Using TD\\_ANYTYPE Parameters](#) and [C Aggregate Function Using TD\\_ANYTYPE Parameters](#).

## Example

The following code excerpt shows an external stored procedure that calls the `FNC_GetAnyTypeParamInfo` function to retrieve information about the `TD_ANYTYPE` arguments.

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xspwanytypefnc(void      *input1,
                    ...,
                    void      *result,
                    int        *indc_input1,
                    ...,
                    int        *indc_result,
                    char       sqlstate[6],
                    SQL_TEXT   extname[129],
                    SQL_TEXT   specific_name[129],
                    SQL_TEXT   error_message[257])
{
    anytype_param_info_t paraminfo[2];
    int numunk;

    FNC_GetAnyTypeParamInfo(2*sizeof(anytype_param_info_t),&numunk,
                           paraminfo);

    if(((paraminfo[1].datatype == CHAR_DT) ||
        (paraminfo[1].datatype == VARCHAR_DT)) &&
        (paraminfo[1].paramIndex == -1)&&
        (paraminfo[1].direction == OUT_PM)&&
        (paraminfo[1].charset == LATIN_CT))
    {
        /* Process the value based upon the data type */
        switch(paraminfo[0].datatype)
        {
            case BYTEINT_DT:
                strcpy ((char *)result,"Byteint");
                break;
                /* Do some processing */
                ...
        }
    }
}

```

The following shows the corresponding SQL procedure definition:

```

CREATE PROCEDURE xspwanytypefnc(IN a TD_ANYTYPE,
                                ...,

```

```

                                OUT result1 TD_ANYTYPE)

LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!xspwanytypefnc!xspwanytypefnc.c';

```

## FNC\_GetAnyTypeInfo\_eon

Returns information about the TD\_ANYTYPE input and output arguments passed to an external routine.

### Syntax

```

void
FNC_GetAnyTypeInfo_eon(int          bufsize,
                       int          *numAnyTypeParams,
                       anytype_param_info_eon_t *AnyTypeAttributeArray);

```

#### **AnyTypeAttributeArray**

Defined in sqltypes\_td.h as:

```

typedef struct anytype_param_info_eon_t {
    INTEGER_td      paramIndex;
    dtype_et        datatype;
    dmode_et        direction;
    INTEGER_td      max_length;
    FNC_LobLength_t lob_length;
    SMALLINT        total_interval_digits;
    SMALLINT        num_fractional_digits;
    charset_et      charset;
    Unicode_Text    UDTName[FNC_MAXNAMELEN_EON];
    SMALLINT        udt_indicator;
} anytype_param_info_eon_t;

```

### Syntax Elements

#### ***bufsize***

Size of the *AnyTypeAttributeArray* buffer.

#### ***numAnyTypeParams***

Number of input and output arguments that is of TD\_ANYTYPE data type.

***AnyTypeAttributeArray***

Pointer to the buffer that will hold the information about the TD\_ANYTYPE parameters.

***AnyTypeAttributeArray***

Pointer to the buffer that will hold the information about the TD\_ANYTYPE parameters.

For more information about dtype\_et, total\_interval\_digits, num\_fractional\_digits, or charset\_et, see [FNC\\_GetStructuredAttributeInfo \[Deprecated\]](#).

For more information about dmode\_et, see [FNC\\_CallSP](#).

***paramIndex***

the parameter positional index starting from 1. For the return parameter, the index will be -1.

***datatype***

the data type of the parameter.

***direction***

indicates whether the parameter is an input, output, or an INOUT parameter.

***max\_length***

the maximum length in bytes that this parameter may utilize. If the parameter is a LOB type, *max\_length* indicates the LOB\_REF length. *lob\_length* provides the length of the LOB data itself.

***lob\_length***

the maximum length of the parameter if it is a LOB type parameter.

***total\_interval\_digits***

the precision value for certain data types. For example, the value *n* in a DECIMAL(*n,m*) type or in INTERVAL DAY(*n*) TO SECOND(*m*). The list of types that use this value is the same as that of attribute\_info\_t.total\_interval\_digits.

***num\_fractional\_digits***

the precision or scale value for certain data types. For example, the value *m* in a DECIMAL(*n,m*) type or in INTERVAL DAY(*n*) TO SECOND(*m*). The list of types that use this value is the same as that of attribute\_info\_t.num\_fractional\_digits.

***charset***

the server character set associated with a character data type.

**UDTName**

[Required if parameter is a UDT, disallowed otherwise]. The name of the UDT.

**udt\_indicator**

Indicates kind of UDT:

<i>udt_indicator</i>	UDT Kind
0	Not a UDT
1	Structured UDT
2	Distinct UDT For distinct types, the data type returns the underlying type of the distinct type UDT.
3	Internal UDT
4	Array Type

## Usage Notes

To get information about the TD\_ANYTYPE input and output arguments passed to an external routine:

1. Declare an integer variable, such as *numAnytypeParams*.
2. Allocate sufficient memory for the *AnyTypeAttributeArray* buffer to hold information for all of the TD\_ANYTYPE input and result parameters declared in your routine. This should be an array of *anytype\_param\_info\_eon\_t* structures, where each element of the array will contain the information for a particular TD\_ANYTYPE parameter.
3. Call *FNC\_GetAnyTypeInfoParamInfo\_eon* and pass in a pointer to the *AnyTypeAttributeArray* buffer along with the size of the buffer, and a pointer to *numAnytypeParams*.

The *FNC\_GetAnyTypeInfoParamInfo\_eon* function fills the *AnyTypeAttributeArray* buffer with information about each TD\_ANYTYPE parameter. The information for the return type of a TD\_ANYTYPE result parameter is contained in the last element of the array. The number of TD\_ANYTYPE parameters is returned in the *numAnyTypeParams* parameter.

---

**Note:**

Be sure to free the memory allocated to the *AnyTypeAttributeArray* buffer when you have completed processing the information.

---

## Example

The following code excerpt is an example of a UDF that calls the *FNC\_GetAnyTypeInfoParamInfo\_eon* function to retrieve information about the TD\_ANYTYPE arguments.



```

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"

void ScalarFncWAnytype(void *input1,
                      ...,
                      void *result,
                      char sqlstate[6])
{
    anytype_param_info_eon_t paraminfo[2];
    FNC_GetAnyTypeParamInfo_eon(2*sizeof(anytype_param_info_eon_t),
    &numAnyType, paraminfo);

    if(((paraminfo[1].datatype == CHAR_DT) ||
        (paraminfo[1].datatype == VARCHAR_DT)) &&
        (paraminfo[1].paramIndex == -1) &&
        (paraminfo[1].direction == OUT_PM) &&
        (paraminfo[1].charset == LATIN_CT))
    {
        /* Process the value based upon the data type */
        switch(paraminfo[0].datatype)
        {
            case BYTEINT_DT:
                strcpy ((char *)result,"Byteint");
                break;
                /* Do some processing */
                ...
        }
    }
}

```

The following shows the corresponding SQL function definition:

```

REPLACE FUNCTION ScalarFncWAnytype ( p1 TD_ANYTYPE, ...)
    RETURNS TD_ANYTYPE
    NO SQL
    PARAMETER STYLE SQL
    RETURNS NULL ON NULL INPUT
    DETERMINISTIC
    LANGUAGE C
    EXTERNAL NAME 'CS!ScalarFncWAnytype!udfsrc/ScalarFncWAnytype.c';

```

## Example

The following code excerpt shows an external stored procedure that calls the FNC\_GetAnyTypeParamInfo\_eon function to retrieve information about the TD\_ANYTYPE arguments.

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xspwanytypefnc(void      *input1,
                    ...,

```

```

        void      *result,
        int        *indc_input1,
        ...,
        int        *indc_result,
        char        sqlstate[6],
        SQL_TEXT   extname[129],
        SQL_TEXT   specific_name[129],
        SQL_TEXT   error_message[257])
{
    anytype_param_info_eon_t paraminfo[2];
    int numunk;

    FNC_GetAnyTypeParamInfo_eon(2*sizeof(anytype_param_info_eon_t),
        &numunk, paraminfo);

    if(((paraminfo[1].datatype == CHAR_DT) ||
        (paraminfo[1].datatype == VARCHAR_DT)) &&
        (paraminfo[1].paramIndex == -1)&&
        (paraminfo[1].direction == OUT_PM)&&
        (paraminfo[1].charset == LATIN_CT))
    {
        /* Process the value based upon the data type */
        switch(paraminfo[0].datatype)
        {
            case BYTEINT_DT:
                strcpy ((char *)result,"Byteint");
                break;
                /* Do some processing */
                ...
        }
    }
}

```

The following shows the corresponding SQL procedure definition:

```

CREATE PROCEDURE xspwanytypefnc(IN a TD_ANYTYPE,
                                ...,
                                OUT result1 TD_ANYTYPE)

LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!xspwanytypefnc!xspwanytypefnc.c';

```

## FNC\_GetArrayElementCount

Get the total number of elements that are present (set to a value, including NULL) for an ARRAY input parameter defined for an external routine.

### Syntax

```
void
FNC_GetArrayElementCount ( ARRAY_HANDLE  aryHandle,
                           int           *presentElts,
                           int           lastPresentElement[ ] )
```

### Syntax Elements

#### *aryHandle*

the handle to an ARRAY type that is defined to be an input parameter to an external routine.

#### *presentElts*

the number of elements in the ARRAY argument that are set to a value, including NULL. An element that is a structured UDT counts as one attribute.

#### *lastPresentElement*

the information for the last present element of the ARRAY value. For a 1-D ARRAY, the index to the last present element is provided as `lastPresentElement[0]`. If the ARRAY type is n-D, then the complete dimension information for this index is placed in cells `lastPresentElement[1]`, `lastPresentElement[2]`, `lastPresentElement[3]` ... `lastPresentElement[FNC_ARRAYMAXDIMENSIONS]` as needed.

### Usage Notes

FNC\_GetArrayElementCount takes an ARRAY handle as input and returns the number of elements in the ARRAY argument that are set to a value, including NULL. It also returns information about the last present element in the ARRAY.

If the ARRAY argument is empty (meaning that zero elements are initialized), FNC\_GetArrayElementCount always returns the first index in the ARRAY type for the *lastPresentElement* parameter.

### Example

The following is an example of using FNC\_GetArrayElementCount to retrieve the number of present elements for a 1-D ARRAY:

```

void getNumAryElts ( ARRAY_HANDLE *phone_ary,
                    INTEGER      *result,
                    char          sqlstate[6])
{
    int presElementsCount;
    int lastElement[1];

    /* Get the number of elements currently stored in the array. */
    FNC_GetArrayElementCount((*phone_ary), &presElementsCount,
                             lastElement);
    *result = presElementsCount;
    ...
}

```

The following example calls FNC\_GetArrayElementCount to retrieve the number of present elements for an n-D ARRAY:

```

void SeismicStations ( ARRAY_HANDLE *seismic_ary,
                      INTEGER      *result,
                      char          sqlstate[6])
{
    int presElementsCount;
    int lastElement[FNC_ARRAYMAXDIMENSIONS];
    /* Get the number of readings(elements) of seismic_ary. */
    FNC_GetArrayElementCount((*seismic_ary), &presElementsCount,
                             lastElement);
    ...
}

```

## FNC\_GetArrayElements

Returns the values of one or more elements from an ARRAY argument and modifies an allocated NullBitVector to set the appropriate bits to indicate whether the elements are present and not NULL, or present but set to NULL.

### Syntax

```

void
FNC_GetArrayElements ( ARRAY_HANDLE    aryHandle,
                      bounds_t        *arrayInterval,
                      void             *returnValue,
                      long             returnValueBufSize,

```

```

NullBitVecType *nullBitVec,
long           nullBitVecBufSize,
long           *length )

```

## Syntax Elements

### *aryHandle*

the handle to an ARRAY type that is defined to be an input parameter to an external routine.

### *arrayInterval*

an array of `bounds_t` structures that provides the index to the set of ARRAY elements that will be retrieved.

For a 1-D ARRAY, the index to the set of elements is provided in the first cell of the *arrayInterval* array. If the ARRAY type is n-D, then subsequent dimension information is placed in cells 2-5 as needed.

The `bounds_t` structure is defined in `sqltypes_td.h` as:

```

typedef struct bounds_t {
    int lowerBound;
    int upperBound;
} bounds_t;

```

The value of *lowerBound* specifies the first value in the given dimension to be retrieved, and *upperBound* specifies the last value in the given dimension to be retrieved.

This range of elements must be sequential for a 1-D ARRAY, and must be specified as a slice or rectangle of elements for an n-D ARRAY. The boundaries (lower and upper) for each dimension must be within the limits defined for the ARRAY type.

### *returnValue*

a pointer to a buffer that `FNC_GetArrayElements` uses to return the values of the elements. The elements are returned in the desired range specified by *arrayInterval* in row-major order.

For each element returned, the *returnValue* buffer contains the following information:

The first 4 bytes describe the size of the data. This is only applicable for variable-length element data types.

The remaining bytes allocated for each element are allocated to hold the maximum size of the element data type. This space contains the actual element value.

Therefore, for each element, space is allocated as:

- (MAX\_SIZE\_OF\_ELEMENT\_DATA\_TYPE + 4 bytes) for variable-length element data types.

- (MAX\_SIZE\_OF\_ELEMENT\_DATA\_TYPE) for fixed-length data types.

***returnValueBufSize***

the size in bytes that was allocated to the *returnValue* argument.

***nullBitVec***

a pointer to a NullBitVector array previously allocated by the caller. For the range of elements requested, the relevant bits of *nullBitVec* will be set to:

- 1 if the element is present and non-null.
- 0 if the element is present but set to NULL.

***nullBitVecBufSize***

the size of the NullBitVector as allocated by the caller prior to initialization of the NullBitVector by setting all bytes to 0.

This parameter is required for protected mode execution of FNC\_GetArrayElements.

***length***

the size in bytes of the value that FNC\_GetArrayElements returns in *returnValue*.

For character data types, the length includes the size of any null termination characters.

## Usage Notes

FNC\_GetArrayElements takes *ary\_handle*, *arrayInterval*, *returnValueBufSize*, and *nullBitVecBufSize* as input arguments and returns:

- The values of the requested elements, specified by *arrayInterval*, in row-major order. The values are placed in the *returnValue* buffer.
- A NullBitVector where the bits of *nullBitVec* that correspond to the range of requested elements are set to indicate whether the elements are present and not NULL, or present but set to NULL.
- The size in bytes of the value that is returned in *returnValue*. The size is placed in the *length* output parameter.

The range of elements requested starts with the first element specified in the *arrayInterval* string and ends with the last element specified. Any elements that are NULL will have space returned which is of the size of the element type, but will be zero-filled. They will also have their presence bit set to 0 in *nullBitVec*. If any element in the requested range is uninitialized, an error is returned to the user.

You can check the value of the relevant presence bits in *nullBitVec* by calling FNC\_CheckNullBitVector or FNC\_CheckNullBitVectorByElemIndex. You can then use this information to interpret the elements returned by FNC\_GetArrayElements.

For more information, see [FNC\\_CheckNullBitVector](#), [FNC\\_CheckNullBitVectorByElemIndex](#), and [Accessing the Value of Array Elements](#).

## Allocating the *returnValue* Buffer

Before calling `FNC_GetArrayElements`, you must allocate the buffer pointed to by *returnValue* using the C data type that maps to the underlying type of the element. Be sure to allocate the buffer for the number of elements which has been requested. For example, if the underlying type of the element is an SQL INTEGER data type, declare the buffer like this:

```
INTEGER value;
```

If the element type is a distinct type, allocate a buffer using the C data type that the distinct type represents. For example, if the element is a distinct type that represents an SQL SMALLINT data type, declare the buffer like this:

```
SMALLINT value;
```

If the underlying type of the element is a character string, `FNC_GetArrayElements` returns a null-terminated character string. The buffer you define must be large enough to accommodate null termination.

If the element type is a structured type, `FNC_GetArrayElements` returns a UDT handle. To get descriptive information on the attributes of the structured type, pass the UDT handle to `FNC_GetStructuredAttributeInfo`. For more information, see [FNC\\_GetStructuredAttributeInfo \[Deprecated\]](#).

To guarantee that the value you pass in for the *returnValueBufSize* argument matches the length of the data type, use the data type length macros defined in `sqltypes_td.h`. For UDT element types, use the macro `SIZEOF_UDT_HANDLE` in calculating the total size of the *returnValue* buffer since UDT handles are returned. For a list of the data type length macros, see [FNC\\_GetStructuredAttribute](#).

Be sure to release allocated resources once you have processed the data.

## Restrictions

If the element type is a UDT, then UDT handles are returned in the *returnValue* buffer for each UDT element retrieved. No more than 2000 UDT handles may be allocated during the execution of any UDF, UDM, or external stored procedure. Therefore, no more than 2000 UDT elements may be retrieved in any call to `FNC_GetArrayElements`, and if other FNC calls in your routine also allocate UDT handles, this limit will be even lower.

## Example

This example is based on the following n-D ARRAY definition:

```

/*Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE myArray AS VARRAY(1:20)(1:20) OF integer;
CREATE TYPE myArray AS integer ARRAY[1:20][1:20];

/*This function returns a range of element values */
void getMultiElement ( ARRAY_HANDLE *ary_handle,
                      void          *result,
                      char          sqlstate[6])
{
    long length;
    long bufferSize = sizeof_integer * 4;
    int *resultBuf;
    NullBitVecType *NullBitVector;
    array_info_t arrayInfo;
    int totalElements;
    long nullVecBufSize;
    bounds_t *arrayRange;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*ary_handle,
                        &arrayInfo,
                        arrayScope);
    nullVecBufSize = arrayInfo.totalNumElements;

    resultBuf = (int*)FNC_malloc(bufferSize);

    arrayRange =
    (bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

    /* Allocate a new NullBitVector to pass to FNC_GetArrayElements */
    NullBitVector = (NullBitVecType*)FNC_malloc(nullVecBufSize);

    /* Set all members of NullBitVector to 0 */
    memset(NullBitVector, 0, nullVecBufSize);

    /* Set values of arrayRange to correspond to the range [1:2][2:3] */
    arrayRange[0].lowerBound = 1;
    arrayRange[0].upperBound = 2;
    arrayRange[1].lowerBound = 2;
    arrayRange[1].upperBound = 3;

    /* Get elements within the range [1:2][2:3] of myArray. */
    FNC_GetArrayElements(*ary_handle, &arrayRange, resultBuf,
                        bufferSize, NullBitVector, NullVecBufSize, &length);
}

```

This example results in an array of 4 integers contained in *resultBuf*, where the 4 values are:

```
10 12 20 33
```

These values may be accessed directly by retrieving the appropriate entry in *resultBuf* as an integer value.

## FNC\_GetArrayNumDimensions

Returns the number of dimensions defined for a given ARRAY type.



## Syntax

```
void
FNC_GetArrayNumDimensions ( ARRAY_HANDLE   aryHandle,
                           int             *numDimensions);
```

### Syntax Elements

#### *aryHandle*

the handle to an ARRAY type that is defined to be an input parameter to an external routine.

#### *numDimensions*

the number of dimensions the ARRAY argument is defined with.

## Usage Notes

FNC\_GetArrayNumDimensions takes an ARRAY handle as input and returns the number of dimensions defined for the ARRAY in the *numDimensions* output parameter.

You can use this function to retrieve the number of dimensions in an ARRAY type when this information is not known to the external routine. This also enables the UDF writer to write code so that it is dimension independent for any ARRAY types accessed in the routine.

## Example

The following is an example of using FNC\_GetArrayNumDimensions to retrieve the number of dimensions in an ARRAY type.

This example is based on the following n-D ARRAY definition:

```
CREATE TYPE myArray AS INTEGER ARRAY[1:20][1:20];

void getNumAryDims ( ARRAY_HANDLE   *phone_ary,
                    INTEGER         *result,
                    char             sqlstate[6])
{
    int numDimensions;
    /* Get the number of dimensions the array is defined with. */
    FNC_GetArrayNumDimensions((*phone_ary), &numDimensions);
    *result = numDimensions;
    ...
}
```

## FNC\_GetArrayTypeInfo [Deprecated]

This function is deprecated because it truncates object names to 30 characters. However, it remains available to support legacy applications. For current and future development, use the corresponding function that includes "EON" in the function name. For example, use FNC\_DbsInfo\_EON instead of FNC\_DbsInfo.

Get information such as the element data type, number of dimensions, total number of elements, and scope information about an ARRAY input parameter defined for an external routine.

### Syntax

```
void
FNC_GetArrayTypeInfo ( ARRAY_HANDLE   aryHandle,
                      array_info_t   *arrayInfo,
                      bounds_t       *arrayScope)
```

#### ***arrayInfo***

Defined in sqltypes\_td.h as:

```
typedef struct array_info_t {
    int numDimensions;
    int totalNumElements;
    element_info_t elementInfo;
} array_info_t;
```

#### ***arrayScope***

Defined in sqltypes\_td.h as:

```
typedef struct bounds_t {
    int lowerBound;
    int upperBound;
} bounds_t;
```

You must declare the *arrayScope* array as follows:

```
bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];
```

#### ***elementInfo***

Defined in sqltypes\_td.h as:

```
typedef struct element_info_t {
    dtype_et data_type;
```

```

    SMALLINT udt_indicator;
    [ CHARACTER udt_type_name[256]; ]
    INTEGER max_length;
    SMALLINT total_interval_digits;
    SMALLINT num_fractional_digits;
    charset_et charset_code;
} element_info_t;

```

## Syntax Elements

### ***aryHandle***

The handle to an ARRAY type that is defined to be an input parameter to an external routine.

### ***arrayInfo***

Pointer to the buffer that will hold the information about the ARRAY input parameter.

For more information about dtype\_et, total\_interval\_digits, num\_fractional\_digits, or charset\_et, see [FNC\\_GetStructuredAttributeInfo \[Deprecated\]](#).

### ***arrayScope***

An array of bounds\_t structures that describes the scope of each dimension in an n-D ARRAY. If the ARRAY type is 1-D, then the default scope information for that single dimension (beginning with 1, ending with *n*, where *n* is the size of the ARRAY) is provided in the first cell of the *arrayScope* array. If the ARRAY type is n-D, then subsequent dimension information is placed in cells 2 to 5 as needed, and provides the beginning and ending bound for each dimension.

The array has a size of FNC\_ARRAYMAXDIMENSIONS, which is set to the value 5 to indicate that an ARRAY may have a maximum of 5 dimensions.

### ***numDimensions***

Number of dimensions defined for the ARRAY type.

### ***totalNumElements***

Total number of elements (across all dimensions) defined for the ARRAY type. This is also known as the maximum cardinality of the ARRAY type.

### ***lowerBound***

Lower bound of a dimension.

***upperBound***

Upper bound of a dimension.

***data\_type***

Data type of the elements of the ARRAY.

***udt\_indicator***

Indicates kind of UDT:

<i>udt_indicator</i>	UDT Kind
0	Not a UDT
1	Structured UDT
2	Distinct UDT For distinct types, the data type returns the underlying type of the distinct type UDT.
3	Internal UDT
4	Array Type

***udt\_type\_name***

[Required if parameter is a UDT, disallowed otherwise]. UDT type name associated with the element.

***max\_length***

Maximum length in bytes of an element value.

You can use *max\_length* as the size in bytes of the buffer you need to allocate before you make a call to FNC\_GetArrayElement to get the values of one element of an ARRAY.

***total\_interval\_digits***

Precision value for certain element types. For example, the value *n* in a DECIMAL(*n,m*) type or in INTERVAL DAY(*n*) TO SECOND(*m*). The list of types that use this value is the same as that of attribute\_info\_t.total\_interval\_digits.

***num\_fractional\_digits***

Precision or scale value for certain element types. For example, the value *m* in a DECIMAL(*n,m*) type or in INTERVAL DAY(*n*) TO SECOND(*m*). The list of types that use this value is the same as that of attribute\_info\_t.num\_fractional\_digits.

***charset\_code***

Server character set associated with the element if the element is a character type.

## Usage Notes

You can use `FNC_GetArrayTypeInfo` but it truncates a retrieved object name after 30 characters. Use `FNC_GetArrayTypeInfo_EON` to avoid truncation of longer object names.

`FNC_GetArrayTypeInfo` takes an `ARRAY` handle as input and returns information about the `ARRAY` argument as follows:

- The number of dimensions, total number of elements, and information about the element data type are returned in the *arrayInfo* buffer.
- The scope of each dimension of the `ARRAY` is returned in the *arrayScope* output parameter.

You must allocate the *arrayInfo* buffer before calling `FNC_GetArrayTypeInfo`. Be sure to free the memory allocated to the buffer after you process the information.

## Example

```
void getArrayElements ( ARRAY_HANDLE  *ary_handle,
                        INTEGER        *result,
                        char            sqlstate[6])
{
    array_info_t arrayInfo;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];
    int bufSize;
    int numDim;

    /* Get element type information for the input ARRAY */
    FNC_GetArrayTypeInfo(*ary_handle,
                        &arrayInfo,
                        arrayScope);
    bufSize = arrayInfo.elementInfo.max_length;
    numDim = arrayInfo.numDimensions;
    ...
}
```

## FNC\_GetArrayTypeInfo\_EON

Get information such as the element data type, number of dimensions, total number of elements, and scope information about an `ARRAY` input parameter defined for an external routine.

## Syntax

```
void
FNC_GetArrayTypeInfo_EON ( ARRAY_HANDLE      aryHandle,
                           array_info_eon_t  *arrayInfo,
                           bounds_t          *arrayScope)
```

### ***arrayInfo***

Defined in `sqltypes_td.h` as:

```
typedef struct array_info_eon_t {
    int numDimensions;
    int totalNumElements;
    element_info_eon_t elementInfo;
} array_info_eon_t;
```

### ***arrayScope***

Defined in `sqltypes_td.h` as:

```
typedef struct bounds_t {
    int lowerBound;
    int upperBound;
} bounds_t;
```

You must declare the *arrayScope* array as follows:

```
bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];
```

### ***elementInfo***

Defined in `sqltypes_td.h` as:

```
typedef struct element_info_eon_t {
    dtype_et data_type;
    SMALLINT udt_indicator;
    [ CHARACTER udt_type_name[FNC_MAXNAMELEN_EON]; ]
    INTEGER_td max_length;
    SMALLINT total_interval_digits;
    SMALLINT num_fractional_digits;
    charset_et charset_code;
} element_info_eon_t;
```

## Syntax Elements

### *aryHandle*

The handle to an ARRAY type that is defined to be an input parameter to an external routine.

### *arrayInfo*

Pointer to the buffer that will hold the information about the ARRAY input parameter.

For more information about `dtype_et`, `total_interval_digits`, `num_fractional_digits`, or `charset_et`, see [FNC\\_GetStructuredAttributeInfo\\_EON](#).

### *arrayScope*

An array of `bounds_t` structures that describes the scope of each dimension in an n-D ARRAY. If the ARRAY type is 1-D, then the default scope information for that single dimension (beginning with 1, ending with *n*, where *n* is the size of the ARRAY) is provided in the first cell of the *arrayScope* array. If the ARRAY type is n-D, then subsequent dimension information is placed in cells 2 to 5 as needed, and provides the beginning and ending bound for each dimension.

The array has a size of `FNC_ARRAYMAXDIMENSIONS`, which is set to the value 5 to indicate that an ARRAY may have a maximum of 5 dimensions.

### *numDimensions*

Number of dimensions defined for the ARRAY type.

### *totalNumElements*

Total number of elements (across all dimensions) defined for the ARRAY type. This is also known as the maximum cardinality of the ARRAY type.

### *lowerBound*

Lower bound of a dimension.

### *upperBound*

Upper bound of a dimension.

### *data\_type*

Data type of the elements of the ARRAY.

### *udt\_indicator*

Indicates kind of UDT:

<i>udt_indicator</i>	UDT Kind
0	Not a UDT
1	Structured UDT
2	Distinct UDT For distinct types, the data type returns the underlying type of the distinct type UDT.
3	Internal UDT
4	Array Type

***udt\_type\_name***

[Required if parameter is a UDT, disallowed otherwise]. UDT type name associated with the element.

***max\_length***

Maximum length in bytes of an element value.

You can use *max\_length* as the size in bytes of the buffer you need to allocate before you make a call to `FNC_GetArrayElement` to get the values of one element of an ARRAY.

***total\_interval\_digits***

Precision value for certain element types. For example, the value *n* in a `DECIMAL(n,m)` type or in `INTERVAL DAY(n) TO SECOND(m)`. The list of types that use this value is the same as that of `attribute_info_t.total_interval_digits`.

***num\_fractional\_digits***

Precision or scale value for certain element types. For example, the value *m* in a `DECIMAL(n,m)` type or in `INTERVAL DAY(n) TO SECOND(m)`. The list of types that use this value is the same as that of `attribute_info_t.num_fractional_digits`.

***charset\_code***

Server character set associated with the element if the element is a character type.

## Usage Notes

`FNC_GetArrayTypeInfo_EON` takes an ARRAY handle as input and returns information about the ARRAY argument as follows:

- The number of dimensions, total number of elements, and information about the element data type are returned in the *arrayInfo* buffer.



- The scope of each dimension of the ARRAY is returned in the *arrayScope* output parameter.

You must allocate the *arrayInfo* buffer before calling `FNC_GetArrayTypeInfo_EON`. Be sure to free the memory allocated to the buffer after you process the information.

## FNC\_GetByteLength

Returns the length, in bytes, for an external routine input argument that has a data type of BYTE.

The length returned is the length of the BYTE data when the external routine is invoked.

## Syntax

```
int
FNC_GetByteLength(void *inputData);
```

### Syntax Elements

#### *inputData*

a pointer to an external routine input argument that has a data type of BYTE.

## Usage Notes

Undefined results occur if the *inputData* argument points to data that is not BYTE.

## Example

```
#define SQL_TEXT Latin_Text
#include <string.h>
#include "sqltypes_td.h"

void strrevcomp_byte(BYTE *a, VARBYTE *result)
{
    int count;
    int i=0;

    count = FNC_GetByteLength(a);
    while(i<count)
    {
        result->bytes[i]=a[i];
        i++;
    }
}
```

```

    result->length=count;
}

```

## FNC\_GetCharLength

Returns the length, in bytes, for an external routine input argument that has a CHAR data type.

The length returned is the length of the CHAR data when the external routine is invoked.

If the server character set of the CHAR data is UNICODE, the value returned is the number of characters multiplied by two.

## Syntax

```

int
FNC_GetCharLength(void *inputString);

```

### Syntax Elements

#### *inputString*

a pointer to an external routine input argument that has a data type of CHAR.

## Usage Notes

You can use *FNC\_GetCharLength* instead of *strlen* to get the length of the input string.

Undefined results occur if the *inputString* argument points to data that is not CHAR.

A UDF that is used for the algorithmic compression of VARCHAR or CHAR columns must call *FNC\_GetCharLength* to get the actual length of the input string.

## Example

```

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"

void strrev12(Character_Latin *input,
             int             inplen,
             VARBYTE         *result)
{
    int i;
    for (i=0; i< inplen; i++)
        result->bytes[inplen-i-1] = input[i];
    result->length = inplen;
}

```

```

void strrevcomp_char(Character_Latin *InputValue,
                    Varbyte      *ResultValue,
                    char          sqlstate[6])
{
    int nullIndicator;
    int length;

    if (InputValue == NULL)
    {
        strcpy(sqlstate, "03288");
        return;
    }
    length = FNC_GetCharLength(InputValue);
    strrev12(InputValue, length, ResultValue);
    strcpy(sqlstate, "00000");
}

```

## FNC\_GetDatasetInfo

Retrieves information about a particular DATASET data type instance, such as the maximum length, inline length, and storage format.

### Syntax

```

void
FNC_GetDatasetInfo ( DATASET_HANDLE      datasetHandle,
                    int*                 maxLength,
                    int*                 inLineLength,
                    int*                 schemaLength,
                    int*                 rawDataLength,
                    dataset_storage_et* storageFormat,
                    boolean_t*          dataLob,
                    boolean_t*          schemaLob)

```

### Syntax Elements

#### *datasetHandle*

A handle to a DATASET data type instance that is defined to be an input parameter to an external routine.

#### *maxLength*

Return parameter.

Specifies the maximum possible length of this DATASET data type instance in bytes.

#### ***inLineLength***

Return parameter.

Specifies the inline length of this DATASET data type instance in bytes.

#### ***schemaLength***

Return parameter.

Specifies the length of the schema for this DATASET data type instance in terms of the number of bytes in the UTF-8 encoding.

#### ***rawDataLength***

Return parameter.

Specifies the length of the data for this DATASET data type instance.

#### ***storageFormat***

Return parameter.

Specifies the storage format of this DATASET data type instance based on the `dataset_storage_et` enumerated type defined in `sqltypes_td.h`.

#### ***dataLob***

Return parameter.

If this DATASET type instance stores its value as a LOB, this will be set to 1. Otherwise this value will be set to 0.

#### ***schemaLob***

Return parameter.

If this DATASET type instance stores its schema as a LOB, this will be set to 1. Otherwise this value will be set to 0.

## **Example: FNC\_GetDatasetInfo**

### **Example Setup**

This example references the following table and data.

```
CREATE TABLE datasetTable(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

avro01.data:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002 |1|base16
```

```
.import vartext file avro01.data
USING (avroData VARCHAR(1000), id varchar(10), encoding VARCHAR(20))
INSERT INTO datasetTable (cast(:id AS
INTEGER),cast(TO_BYTES(:avroData, :encoding) AS DATASET STORAGE FORMAT AVRO));
```

### Example Using FNC\_GetDatasetInfo

This example uses FNC\_GetDatasetInfo to retrieve information about a DATASET data type instance.

```
CREATE FUNCTION GetDatasetInfo ( a1 TD_ANYTYPE)
RETURNS VARCHAR(1000)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!GetDatasetInfo!GetDatasetInfo.c!F!GetDatasetInfo';
```

GetDatasetInfo.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>

#define buffer_size 512

void GetDatasetInfo(DATASET_HANDLE *input,
                    VARCHAR_LATIN *result,
                    int *inputNullIndicator,
                    int *outputNullIndicator,
                    char sqlstate[6],
                    SQL_TEXT extname[129],
                    SQL_TEXT specific_name[129],
                    SQL_TEXT error_message[257])
```

```

{
    int maxLength, inlineLength, schemaLength, rawDataLength = 0;
    dataset_storage_et storageFormat = 0;
    boolean_t dataLob, schemaLob = 0;

    FNC_GetDatasetInfo(*input, &maxLength, &inlineLength, &schemaLength,
&rawDataLength, &storageFormat, &dataLob, &schemaLob);

    sprintf(result, "Max: %d, Inline: %d, SchemaLen: %d, RawDataLen: %d,
Storage: %s, DataLob: %s, SchemaLob: %s\0",
            maxLength, inlineLength, schemaLength, rawDataLength,
            (storageFormat == DATASET_Avro_EN ? "Avro" : "Unknown"),
            (dataLob == 1 ? "Yes" : "No"),
            (schemaLob == 1 ? "Yes" : "No"));

    sprintf(sqlstate, "00000\0");
    *outputNullIndicator = 0;
}

```

The following is sample output:

```

SELECT GetDatasetInfo(avroFile) FROM datasetTable;

> Max: 2097088000, Inline: 10000, SchemaLen: 70, RawDataLen: 1, Storage: Avro,
DataLob: No, SchemaLob: No

```

## FNC\_GetDatasetInputLob

This routine allows users to read DATASET data that is stored as a LOB using the LOB FNC routines.

### Syntax

```

void
FNC_GetDatasetInputLob( DATASET_HANDLE    datasetHandle,
                        LOB_LOCATOR*      instance )

```

### Syntax Elements

#### *datasetHandle*

A handle to a DATASET data type instance that is defined to be an input parameter to an external routine.

**instance**

A pointer to a LOB\_LOCATOR which will be used to read the LOB data of a DATASET instance.

**Usage Notes**

Use FNC\_GetDatasetInputLob only when the DATASET data is stored as a LOB. If FNC\_GetDatasetInfo returns dataLob > 0, you can use FNC\_GetDatasetInputLob; otherwise, you should use FNC\_GetInternalValue instead.

For Avro instances, this routine returns a LOB\_LOCATOR which allows a user to access a buffer that contains the UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value. This is equivalent to the transform format.

If the DATASET data is a CSV value, the value does not include any optional schema.

**Example: FNC\_GetDatasetInputLob****Example Setup**

This example references the following table and data.

```
CREATE TABLE datasetTable(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

avro01.data:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002 |1|base16
```

```
.import vartext file avro01.data
USING (avroData VARCHAR(1000), id varchar(10), encoding VARCHAR(20))
INSERT INTO datasetTable (cast(:id AS
INTEGER),cast(TO_BYTES(:avroData, :encoding) AS DATASET STORAGE FORMAT AVRO));
```

**Example Using FNC\_GetDatasetInputLob**

```
REPLACE FUNCTION getLobData ( a1 TD_ANYTYPE)
RETURNS TD_ANYTYPE
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getLobData!getLobData.c!F!getLobData';
```

getLobData.c:

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
void getLobData ( DATASET_HANDLE *dataset_instance,
                  void *result,
                  char sqlstate[6])
{
    int max_length = 0;
    int in_line_length = 0;
    int schema_length = 0;
        int raw_data_length=0;
        dataset_storage_et dataset_storage = DATASET_INVALID_EN;
    boolean_t dataLob = 0;
    boolean_t schemaLob = 0;

    /* Get the info of the DATASET instance. */
    FNC_GetDatasetInfo((*dataset_instance), &max_length, &in_line_length,
&schema_length, &raw_data_length, &dataset_storage, &dataLob, &schemaLob);

    if (dataLob == 1)
        {
            LOB_LOCATOR inLOB;
            LOB_CONTEXT_ID id;
            FNC_LobLength_t readlen, actualInputLength;
            int trunc_err = 0;
            BYTE *bufPtr = result;
            int buffer_size = 0;

                FNC_GetDatasetInputLob((int)*dataset_instance, &inLOB);
            FNC_LobOpen(inLOB, &id, 0, 0);
            buffer_size = FNC_GetLobLength(inLOB);

            while (FNC_LobRead(id, bufPtr, buffer_size, &readlen) == 0 && !trunc_err)
            {
                bufPtr += readlen;
                memcpy(result,bufPtr,readlen);
            }
            FNC_LobClose(id);

```



```
    }
}
```

The following is sample output:

```
/*if data is not a LOB, there will be no output*/

SELECT (getLobData (avroFile) RETURNS VARBYTE(1000)) FROM datasetTable;
>
```

## FNC\_GetDatasetResultLob

This routine allows DATASET data to be written to a LOB associated with a DATASET instance.

### Syntax

```
void
FNC_GetDatasetResultLob( DATASET_HANDLE      datasetHandle,
                        LOB_RESULT_LOCATOR* instance  )
```

### Syntax Elements

#### *datasetHandle*

A handle to a DATASET data type instance that is defined to be an input parameter to an external routine.

#### *instance*

A pointer to a LOB\_RESULT\_LOCATOR to be used to write the LOB data of a DATASET instance.

### Usage Notes

Use FNC\_GetDatasetResultLob only when the DATASET data will be stored as a LOB. If FNC\_GetDatasetInfo returns dataLob > 0, you can use FNC\_GetDatasetResultLob; otherwise, you should use FNC\_SetInternalValue instead.

The LOB\_RESULT\_LOCATOR obtained may be used with all LOB FNC routines that allow the user to write to this LOB.

For Avro instances, this routine returns a LOB\_RESULT\_LOCATOR which allows a user to write the UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value to a buffer. This is equivalent to the transform format. Failure to write the data in this format results in an error.

For CSV values, the value is written to the result LOB, but does not include any optional schema.

## Example: FNC\_GetDatasetResultLob and FNC\_SetDatasetLob

### Example Setup

This example references the following table and data.

```
CREATE TABLE datasetTable(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

avro01.data:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002 |1|base16
```

```
.import vartext file avro01.data
USING (avroData VARCHAR(1000), id varchar(10), encoding VARCHAR(20))
INSERT INTO datasetTable (cast(:id AS
INTEGER),cast(TO_BYTES(:avroData, :encoding) AS DATASET STORAGE FORMAT AVRO));
```

### Example Using FNC\_GetDatasetResultLob and FNC\_SetDatasetLob

```
CREATE FUNCTION CreateDATASETlob(a1 TD_ANYTYPE)
RETURNS TD_ANYTYPE
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!CreateDATASETlob!CreateDATASETlob.c!F!CreateDATASETlob';
```

CreateDATASETlob.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>

#define buffer_size 64000

void CreateDATASETlob (DATASET_HANDLE *input,
    DATASET_HANDLE *result,
    int *inputNullIndicator,
    int *outputNullIndicator,
```

```

        char sqlstate[6],
        SQL_TEXT extname[129],
        SQL_TEXT specific_name[129],
        SQL_TEXT error_message[257])

{
    LOB_LOCATOR inLOB;
    LOB_RESULT_LOCATOR outLOB;
    BYTE buffer[buffer_size];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen, actualInputLength;
    int trunc_err = 0;
    FNC_GetDatasetInputLob(*input,&inLOB);
    FNC_GetDatasetResultLob(*result,&outLOB);

    readlen=0;
    writelen=0;
    actualInputLength = 0;

    FNC_LobOpen(inLOB, &id, 0, 0);
    while( FNC_LobRead(id, buffer, buffer_size, &readlen) == 0 && !
trunc_err )
    {
        trunc_err = FNC_LobAppend(outLOB, buffer, readlen, &writelen);
    }
    FNC_LobClose(id);

    FNC_SetDatasetLob(*result,outLOB);
    sprintf(sqlstate, "00000\0");
    *outputNullIndicator = 0;
}

```

The following is sample output that shows the text representation of the data:

```

SELECT (CreatedatasetLob(avroFile) RETURNS DATASET STORAGE FORMAT Avro).toJSON()
FROM datasetTable;

>      {"a":1}

```

## FNC\_GetDatasetSchema

Retrieves the schema for a DATASET data type instance.

## Syntax

```
void
FNC_GetDatasetSchema ( DATASET_HANDLE          datasetHandle,
                       void*                   schemaBuf,
                       int                     schemaBufLen,
                       int*                    actualSchemaLength,
                       dataset_schema_encoding_t schemaEncoding)
```

## Syntax Elements

### *datasetHandle*

A handle to a DATASET data type instance that is defined to be an input parameter to an external routine.

### *schemaBuf*

Return parameter that contains the UNICODE characters in JSON format representing the schema.

### *schemaBufLen*

The length of the buffer used to store the schema, in bytes.

### *actualSchemaLength*

Return parameter specifying the length of the schema for this DATASET data type instance, in bytes.

### *schemaEncoding*

The encoding in which the user would like to retrieve the schema.

## Usage Notes

FNC\_GetDatasetSchema can be used to retrieve the schema of a DATASET data type, regardless of storage format or size. The schema is returned as UNICODE text, encoded in either UTF-8 or UTF-16, depending on the value of the *schemaEncoding* parameter.

Note that the schema is returned without any special formatting, in its JSON encoding and with UNICODE characters.

Use FNC\_GetDatasetInfo to determine the appropriate buffer size that can store the schema. If the buffer passed into this routine is too small to contain the entire schema, an error is reported.

For best performance, Use FNC\_GetDatasetSchema when the schema is *not* stored as a LOB. If FNC\_GetDatasetInfo returns schemaLob = 0, you can use FNC\_GetDatasetSchema; otherwise, you should use FNC\_GetDatasetSchemaLob instead.

## Example: FNC\_GetDatasetSchema

### Example Setup

This example references the following table and data.

```
CREATE TABLE datasetTable(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

avro01.data:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002 |1|base16
```

```
.import vartext file avro01.data
USING (avroData VARCHAR(1000), id varchar(10), encoding VARCHAR(20))
INSERT INTO datasetTable (cast(:id AS
INTEGER),cast(TO_BYTES(:avroData, :encoding) AS DATASET STORAGE FORMAT AVRO));
```

### Example Using FNC\_GetDatasetSchema

```
CREATE FUNCTION getSchema ( a1 TD_ANYTYPE, a2 integer)
RETURNS TD_ANYTYPE
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getSchema!getSchema.c!F!getSchema';
```

getSchema.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>

#define BUFFER_SIZE 10000

void getSchema(void *input,
               INTEGER *schemaEncoding,
               LOB_RESULT_LOCATOR *result,
               int *input1NullIndicator,
```

```

        int *input2NullIndicator,
        int *outputNullIndicator,
        char sqlstate[6],
        SQL_TEXT extname[129],
        SQL_TEXT specific_name[129],
        SQL_TEXT error_message[257])
{
    void* schemaBuf;
    int schemaBufLen = 0;
    int actualSchemaLength = 0;
    FNC_LobLength_t writeLen = 0;
    int maxLength, inlineLength, schemaLength, rawDataLength = 0;
    dataset_storage_et storageFormat = 0;
    boolean_t dataLob, schemaLob = 0;

    FNC_GetDatasetInfo(*((DATASET_HANDLE*)(input)), &maxLength, &inlineLength,
&schemaLength, &rawDataLength, &storageFormat, &dataLob, &schemaLob);

    if (*schemaEncoding == datasetSchemaUTF8)
    {
        schemaBufLen = schemaLength;
    }
    else
    {
        schemaBufLen = (schemaLength*2); /*worst case is each UTF8 character is
an ascii character, which will require 2 bytes in UTF16*/;
    }

    schemaBuf = FNC_malloc((size_t)schemaBufLen);

    FNC_GetDatasetSchema(*((DATASET_HANDLE*)(input)),(void*)
schemaBuf,schemaBufLen,&actualSchemaLength,*schemaEncoding);
    FNC_LobAppend(*result,(BYTE*)schemaBuf,
(FNC_LobLength_t)actualSchemaLength,&writeLen);

    FNC_free(schemaBuf);

    sprintf(sqlstate, "00000\0");
    *outputNullIndicator = 0;
}

```

The following is sample output:

```
SELECT from_bytes((GetSchema(avroFile,0) RETURNS BLOB),'ascii')
FROM datasetTable;

> {"type":"record","name":"rec_0","fields":[{"name":"a","type":"int"}]}
```

## FNC\_GetDatasetSchemaLob

Allows users to read the schema of a DATASET data type instance, which is stored as a LOB, using the LOB FNC routines.

### Syntax

```
Void
FNC_GetDatasetSchemaLob( DATASET_HANDLE    datasetHandle,
                        LOB_LOCATOR*       schemaLoc,
                        dataset_schema_encoding_t schemaEncoding)
```

### Syntax Elements

#### *datasetHandle*

A handle to a DATASET type instance that is defined to be an input parameter to an external routine.

#### *schemaLoc*

A pointer to a LOB\_LOCATOR that will be used to read the LOB schema of a DATASET type instance.

#### *schemaEncoding*

The encoding in which the user would like to retrieve the schema.

### Usage Notes

The schema is returned as UNICODE text, encoded in either UTF-8 or UTF-16, depending on the value of the *schemaEncoding* parameter.

Use FNC\_GetDatasetSchemaLob when the schema is stored as a LOB. If FNC\_GetDatasetInfo returns *schemaLob* > 0, you can use FNC\_GetDatasetSchemaLob; otherwise, you should use FNC\_GetDatasetSchema instead.

## Example: FNC\_GetDatasetSchemaLob

### Example Setup

This example references the following table and data.

```
CREATE TABLE datasetTable(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

avro01.data:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002 |1|base16
```

```
.import vartext file avro01.data
USING (avroData VARCHAR(1000), id varchar(10), encoding VARCHAR(20))
INSERT INTO datasetTable (cast(:id AS
INTEGER),cast(TO_BYTES(:avroData, :encoding) AS DATASET STORAGE FORMAT AVRO));
```

### Example Using FNC\_GetDatasetSchemaLob

```
CREATE FUNCTION getSchemaLob ( a1 TD_ANYTYPE, a2 TD_ANYTYPE)
RETURNS TD_ANYTYPE
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getSchemaLob!getSchemaLob.c!F!getSchemaLob';
```

getSchemaLob.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>

#define BUFFER_SIZE 10000

void getSchemaLob(DATASET_HANDLE *input,
    INTEGER *schemaEncoding,
    LOB_RESULT_LOCATOR *result,
    int *input1NullIndicator,
    int *input2NullIndicator,
    int *outputNullIndicator,
    char sqlstate[6],
    SQL_TEXT extname[129],
    SQL_TEXT specific_name[129],
    SQL_TEXT error_message[257])
```



```

{
    BYTE schemaBuf[BUFFER_SIZE] = {'\0'};
    int schemaBufLen = BUFFER_SIZE;
    LOB_LOCATOR schemaInputLob;
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen, actualSchemaLength;
    int trunc_err = 0;
    BYTE nullTerminator = '\0';

    FNC_GetDatasetSchemaLob(*input,&schemaInputLob,*schemaEncoding);

    readlen=0;
    writelen=0;
    actualSchemaLength = 0;

    FNC_LobOpen(schemaInputLob, &id, 0, 0);

    while( FNC_LobRead(id, schemaBuf, schemaBufLen, &readlen) == 0 && !
trunc_err )
    {
        trunc_err = FNC_LobAppend(*result, schemaBuf, readlen, &writelen);
    }
    FNC_LobClose(id);

    sprintf(sqlstate, "00000\0");
    *outputNullIndicator = 0;
}

```

The following is sample output:

```

SELECT from_bytes((GetSchemaLob(avroFile,0) RETURNS BLOB),'ascii')
FROM datasetTable;

> {"type":"record","name":"rec_0","fields":[{"name":"a","type":"int"}]}

```

## FNC\_GetDistinctInputLob

Returns the LOB locator of a distinct type that represents a LOB and is defined to be an input parameter to a UDF, UDM, or external stored procedure.

## Syntax

```
void
FNC_GetDistinctInputLob ( UDT_HANDLE   udtHandle,
                          LOB_LOCATOR *object )
```

### Syntax Elements

#### *udtHandle*

the handle to a distinct UDT that represents a LOB and is defined to be an input parameter to a UDF, UDM, or external stored procedure.

#### *object*

a pointer to the LOB locator represented by *udtHandle*.

## Usage Notes

After you obtain the LOB locator for the distinct type, use the LOB access functions, such as FNC\_LobOpen, to read the data.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_GetDistinctInputLob

```
void textfile_t_toUpperCase( UDT_HANDLE *textfileUdt,
                             UDT_HANDLE *resulttextfileUdt,
                             char        sqlstate[6])
{
    LOB_LOCATOR inDocLoc;

    /* Get a LOB_LOCATOR for the input document */
    FNC_GetDistinctInputLob(*textfileUdt, &inDocLoc);

    ...
}
```

## FNC\_GetDistinctResultLob

Returns the LOB locator of a distinct type that represents a LOB and is defined to be the return value of a UDF or UDM or an INOUT or OUT parameter to an external stored procedure.

### Syntax

```
void
FNC_GetDistinctResultLob ( UDT_HANDLE      udtHandle,
                          LOB_RESULT_LOCATOR *object )
```

### Syntax Elements

#### *udtHandle*

the handle to a distinct UDT that represents a LOB and is defined to be the return value of a UDF or UDM or an INOUT or OUT parameter to an external stored procedure.

#### *object*

a pointer to the LOB locator represented by *udtHandle*.

### Usage Notes

After you obtain the LOB locator for the distinct type, use the LOB access functions, such as FNC\_LobOpen, to append data.

Predefined LOBs and distinct UDTs that represent LOBs behave the same way within UDFs, UDMs, and external stored procedures:

- A LOB result value is empty with a length of zero unless you append data or explicitly set it to null by using the result indicator argument.
- Setting the result indicator argument to -1 discards any data that was appended to the LOB result.
- A distinct UDT that represents a LOB and is defined to be an INOUT parameter to an external stored procedure retains the input version if no data is appended to the LOB.

### Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

### Example Using FNC\_GetDistinctResultLob

```
void textfile_t_toUpperCase( UDT_HANDLE *textfileUdt,
                           UDT_HANDLE *resulttextfileUdt,
```

```

                                char          sqlstate[6])
{
    LOB_RESULT_LOCATOR resultDocLoc;

    /* Get a LOB_RESULT_LOCATOR for the result doc attribute. */
    FNC_GetDistinctResultLob(*resultTextfileUdt, &resultDocLoc);

```

## FNC\_GetDistinctValue

Get the value of a non-LOB distinct type that is defined to be an input parameter to a UDF, UDM, or external stored procedure.

For information on getting the value of a distinct type that represents a LOB, see [FNC\\_GetDistinctInputLob](#).

## Syntax

```

void
FNC_GetDistinctValue ( UDT_HANDLE  udtHandle,
                      void         *returnValue,
                      int          bufSize,
                      int          *length )

```

### Syntax Elements

#### *udtHandle*

the handle to a distinct UDT that is defined to be an input parameter to a UDF, UDM, or external stored procedure.

#### *returnValue*

a pointer to a buffer that FNC\_GetDistinctValue uses to return the value of the distinct type.

#### *bufSize*

the size in bytes of the *returnValue* buffer.

#### *length*

the size in bytes of the value that FNC\_GetDistinctValue returns in *returnValue*.

For character data types, the length includes the size of any null termination characters.

## Usage Notes

Before calling `FNC_GetDistinctValue`, you must allocate the buffer pointed to by *returnValue* using the C data type that maps to the underlying type of the distinct UDT. For example, if the distinct type represents an SQL INTEGER data type, declare the buffer like this:

```
INTEGER value;
```

If the underlying type of the distinct UDT is a character string, `FNC_GetDistinctValue` returns a null-terminated character string. The buffer you define must be large enough to accommodate null termination.

For information on the C data types that you can use, see [C Data Types](#).

To guarantee that the value you pass in for the *bufSize* argument matches the length of the data type, use the following macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_CHARACTER_LATIN_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_KANJISJIS_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_KANJI1_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_UNICODE_WITH_NULL( <i>len</i> )	Returns the length in bytes of a CHARACTER data type of <i>len</i> characters, including null termination characters. For example, the following returns a length of 8 ( $3 * 2 + 2 = 8$ ): <pre>SIZEOF_CHARACTER_UNICODE_WITH_NULL(3)</pre>
SIZEOF_VARCHAR_LATIN_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_KANJISJIS_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_KANJI1_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_UNICODE_WITH_NULL( <i>len</i> )	Returns the length in bytes of a VARCHAR data type of <i>len</i> characters, including null termination characters. For example, the following returns a length of 8 ( $3 * 2 + 2 = 8$ ): <pre>SIZEOF_VARCHAR_UNICODE_WITH_NULL(3)</pre>
SIZEOF_BYTE( <i>len</i> ) SIZEOF_VARBYTE( <i>len</i> )	Returns the length in bytes of the specified BYTE or VARBYTE data type, where <i>len</i> specifies the number of values.
SIZEOF_GRAPHIC( <i>len</i> ) SIZEOF_VARGRAPHIC( <i>len</i> )	Returns the length in bytes of the specified CHARACTER(n) CHARACTER SET GRAPHIC or VARCHAR(n) CHARACTER SET GRAPHIC data type, where <i>len</i> specifies the number of values.
SIZEOF_BYTEINT SIZEOF_SMALLINT SIZEOF_INTEGER SIZEOF_BIGINT SIZEOF_REAL SIZEOF_DOUBLE_PRECISION SIZEOF_FLOAT SIZEOF_DECIMAL1 SIZEOF_DECIMAL2	Returns the length in bytes of the specified numeric data type. For NUMBER, the length returned is $4 + 2 + 17 = 23$ bytes since Vantage allocates max length (17 bytes) for the mantissa.

Macro	Description
SIZEOF_DECIMAL4 SIZEOF_DECIMAL8 SIZEOF_DECIMAL16 SIZEOF_NUMERIC1 SIZEOF_NUMERIC2 SIZEOF_NUMERIC4 SIZEOF_NUMERIC8 SIZEOF_NUMERIC16 SIZEOF_NUMBER	
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.
SIZEOF_INTERVAL_YEAR SIZEOF_IntrvlYtoM SIZEOF_INTERVAL_MONTH SIZEOF_INTERVAL_DAY SIZEOF_IntrvlDtoH SIZEOF_IntrvlDtoM SIZEOF_IntrvlDtoS SIZEOF_HOUR SIZEOF_IntrvlHtoM SIZEOF_IntrvlHtoS SIZEOF_MINUTE SIZEOF_IntrvlMtoS SIZEOF_IntrvlSec	Returns the length in bytes of the specified Interval type.

## Restrictions

An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

## Example Using FNC\_GetDistinctValue

```
void meters_t_toInches( UDT_HANDLE    *metersUdt,
                       FLOAT          *result,
                       char            sqlstate[6])
{
    FLOAT value;
    int length;

    /* Get the value of metersUdt. */
    FNC_GetDistinctValue(*metersUdt, &value, SIZEOF_FLOAT, &length);
}
```

```
...
}
```

## FNC\_GetExtendedJSONInfo

Retrieves the maximum length, inline length, character set, and storage format of a JSON type instance that is used as an input or output parameter in an external routine. The function also indicates whether or not the JSON data is stored as a LOB. This function is similar to FNC\_GetJSONInfo with additional inline length and storage format information being returned.

### Syntax

```
void
FNC_GetExtendedJSONInfo ( JSON_HANDLE      jsonHandle,
                          int               *inlineLength,
                          int               *maxLength,
                          charset_et       *charSet,
                          int               *numLobs,
                          json_storage_et *storageFmt);
```

### Syntax Elements

#### *jsonHandle*

A handle to a JSON type instance that is defined to be a parameter to an external routine.

#### *inlineLength*

Return parameter that specifies the maximum possible size of the JSON instance that can be stored in a row. The size is specified in bytes.

#### *maxLength*

Return parameter that specifies the maximum possible length of the JSON instance in bytes.

#### *charSet*

Return parameter that specifies the character set of the JSON text, either LATIN\_CT or UNICODE\_CT as defined in sqltypes\_td.h. If this is a binary JSON instance, the character set is specified as UNDEF\_CT.

#### *numLobs*

Return parameter that specifies the number of LOBs used to store the JSON data. Valid values are as follows:

- 0, which indicates that the data is not stored as a LOB
- 1, which indicates that the data is stored as a LOB

**storageFmt**

Return parameter that specifies the storage format for the JSON instance. Valid values are defined in `sqltypes_td.h` as follows:

```
typedef enum json_storage_en
{
    JSON_INVALID_EN=-1,
    JSON_TEXT_EN=0,
    JSON_BSON_EN=1,
    JSON_UBJSON_EN=2,
    DATASET_AVRO_EN=3
} json_storage_en;
```

**Example: FNC\_GetExtendedJSONInfo**

```
/*
CREATE FUNCTION MyJSONUDF( a1 JSON(100) )
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!MyJSONUDF!MyJSONUDF.c!F!MyJSONUDF';
*/
```

```
void MyJSONUDF( JSON_HANDLE *json_instance,
                VARCHAR_LATIN *result,
                char          sqlstate[6])
{
    int maxLength = 0;
    int inlineLength = 0;
    charset_et charset = 0;
    int numLobs = 0;
    json_storage_et storageFmt = JSON_INVALID_EN;

    /* Get the info of the JSON instance. */

    FNC_GetExtendedJSONInfo((*json_instance), &inlineLength, &maxLength, &charset,
    &numLobs, &storageFmt);
    sprintf(result, "Inline Length: %d, Max Length: %d, CharSet: %d, NumLobs: %d,
Storage Format: %d", inlineLength, maxLength, charset, numLobs, storageFmt);
```



```
...
}
```

Sample output:

```
> Inline Length: 100, Max Length: 100, CharSet: 1, NumLobs: 0, Storage Format: 0
```

## FNC\_GetGeometryInfo

Returns information about the ST\_Geometry value including its size and whether it stores its value as a LOB.

### Syntax

```
void FNC_GetGeometryInfo(GEO_HANDLE  geoHandle,
                        FNC_GeomSize_t *maxLength,
                        int *numLobs)
```

### Syntax Elements

#### *geoHandle*

A handle to an ST\_Geometry type that is defined to be an input or output parameter to an external routine.

#### *maxLength*

The maximum length possible for this ST\_Geometry value.

#### *numLobs*

If this ST\_Geometry object stores its value as a LOB, *numLobs* is 1, otherwise *numLobs* is 0.

### Usage Notes

FNC\_GetGeomInfo returns information about the ST\_Geometry value such as its maximum size and whether it uses a LOB to store its value. The GEO\_HANDLE for the ST\_Geometry value is passed in as input, and the maximum size and whether it can store its value as a LOB is returned to the caller.

### Example: Using FNC\_GeomGetWKT to retrieve a ST\_Geometry WKT

The following example function takes a ST\_Geometry parameter and retrieves the Well Known Text format for the geometry. It returns the string back in the ST\_Geometry return parameter.

```
CREATE FUNCTION geomUDF1(P1 st_geometry(1000))
RETURNS ST_GEOMETRY(1000)
NO SQL
PARAMETER STYLE SQL
```

```

DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!geomUDF1!geomUDF1.c';

void geomUDF1(GEO_HANDLE* geom_handle,
              GEO_HANDLE* return_handle,
              int* indicator_thisGeom,
              int* indicator_returnGeom,
              char sqlstate[6],
              SQL_TEXT extname[129],
              SQL_TEXT specific_name[129],
              SQL_TEXT error_message[257] )
{
    FNC_GeomSize_t geomsz;
    FNC_GeomSize_t wktsz;
    char* wktBuffer;
    FNC_GeomSize_t wktBufferSize;
    int srid , max_length, numLobs;

    /* Get the Geometry Info */
    FNC_GetGeometryInfo(*geom_handle,&max_length, &numLobs);
    if(numLobs == 0)
    {
        /* Get the Geometry WKT Size */
        geomsz = FNC_GeomGetWKTSz(*geom_handle);

        /* Allocate the buffer to hold the WKT. Note that we add
           1 for the null termination character. */
        wktBuffer = (char*)FNC_Malloc(geomsz + 1);
        wktBufferSize = geomsz + 1;

        /* Get the WKT string */
        FNC_GeomGetWKT(*geom_handle,wktBuffer, wktBufferSize,
                      &wktsz, &srid);

        /* Set the WKT string as return value */
        FNC_GeomSetWKT(*return_handle, wktBuffer, wktsz,0);
        *indicator_returnGeom = 0;
        FNC_free(wktBuffer);
    }
}

```

## FNC\_Get\_GLOP\_Map

Get the GLOP set map information associated with an external routine.

Before an external routine can access any GLOP data, it must call FNC\_Get\_GLOP\_Map. The call serves two purposes:

- It lets the system know that the GLOP memory is going to be used.
- It ensures that new mappings will be registered and that old mappings will be restored.

FNC\_Get\_GLOP\_Map returns an integer that indicates success or failure.

A returned integer of non zero indicates that the mapping has not been done.

Value	Meaning
0	Map was set up.
-1	The GLOP set was not mapped because the maximum vproc GLOP memory limit has been reached. To change the limit, use the <i>cufconfig</i> utility. For details, see <i>Utilities</i> .
-2	The GLOP set was not mapped because the external routine is no longer a member of the GLOP set, the set no longer exists, or the database where the routine resides is no longer a member of the set.

## Syntax

```
int FNC_Get_GLOP_Map (GLOP_Map_t **Map)
```

### Map

```
typedef struct
{
    GLOP_ref_t  GLOP[8],
} GLOP_Map_t;
```

### GLOP\_ref\_t

```
typedef struct
{
    void      *GLOP_ptr,
    int       version,
    int       size,
    int       page,
    GLOP_Mode mode,
```

```
GLOP_Type type,
} GLOP_ref_t;
```

## Syntax Elements

### *Map*

The address of where FNC\_Get\_GLOP\_Map is to store a pointer to a GLOP\_Map\_t structure.

If the external routine has no access to a GLOP set, FNC\_Get\_GLOP\_Map returns a NULL pointer. This is the case when the external routine contains no USING GLOP SET clause in the CREATE statement or the GLOP data table could not find a row for the specified GLOP set of which the external routine is a member.

Structure GLOP\_Map\_t is an array of eight GLOP\_ref\_t structures, plus some internal structures. Eight GLOP data references is the maximum an external routine is allowed to map.

### *GLOP\_ptr*

Specifies the mapped address of the GLOP data.

### *version*

Specifies the version of the GLOP being used.

This version reflects the GLOP\_Version column in the DBCEExtension.GLOP\_Data table.

### *size*

Specifies the length in bytes of the mapped GLOP data.

Referencing outside the range could cause a memory violation.

### *page*

Specifies the number of the mapped GLOP page.

### *mode*

Specifies the mapping and modification mode, where the GLOP\_Mode enumerated type has the following definition:

```
typedef enum
{
    GLOP_NONE=0, /* no map mode */
    GLOP_R=1,    /* read-only */
    GLOP_W=2,    /* normal map read/write */
}
```

```

    GLOP_M=3,      /* normal map global modify */
    GLOP_SW=4,     /* shared map read/write */
    GLOP_SM=5      /* shared map global modify */
} GLOP_Mode;

```

For details on the GLOP mapping and modification modes, see [GLOP Data Attributes](#) and [GLOP Data Attributes](#).

### type

Specifies the type of GLOP, where the GLOP\_Type enumerated type has the following definition:

```

typedef enum
{
    GLOP_NULL=0, /* no mapping */
    GLOP_SY=1,   /* system GLOP */
    GLOP_RO=2,   /* role GLOP */
    GLOP_US=3,   /* user GLOP */
    GLOP_SE=4,   /* session GLOP */
    GLOP_TR=5,   /* transaction GLOP */
    GLOP_RE=6,   /* request GLOP */
    GLOP_XR=7    /* external routine GLOP */
} GLOP_Type;

```

For details on the GLOP types, see [GLOP Types](#).

## Usage Notes

You must call FNC\_Get\_GLOP\_Map first before calling any other GLOP functions such as FNC\_GLOP\_Lock, FNC\_GLOP\_Unlock, FNC\_GLOP\_Map\_Page, or FNC\_GLOP\_Global\_Copy. Also, you should call FNC\_Get\_GLOP\_Map first in each phase of a table function that uses any of the other GLOP routines to initialize the memory pointer for that phase.

When this call completes, the structure is filled in with the address of all GLOP mapped data that the external routine has access to for the particular instance in which it is invoked. If a GLOP\_ptr is NULL then it has no mapping for that particular GLOP data. An external routine cannot assume that the entry is set up. First, it has no way of knowing whether it is supposed to be mapped. Second, it might not be mapped anymore. If the mapping is supposed to exist in order for the external routine to work and it does not exist, the external routine should return with an exception. The way to check is to always reference the mapping like this:

```

#define SYSTEM_MAP  0

Sysinfo_t  SysInfoPtr;

```

```

GLOP_Map_t *MyGLOP;
int          glop_stat;
. . .
glop_stat = FNC_Get_GLOP_Map(&MyGLOP);

if (glop_stat)
    ... process error: Not a member of any GLOP set

if ( MyGLOP->GLOP[SYSTEM_MAP].GLOP_Ptr == NULL)
{
    ... process error: System level map does not exist
};

SysInfoPtr = MyGLOP->GLOP[SYSTEM_MAP].GLOP_ptr;

. . .

```

Here are some rules to consider:

- You can map the same GLOP data in multiple map indexes except for role, user, or external routine GLOP data. That way you can map different pages of the same read-only GLOP data at the same time. This has to be set up that way in the GLOP\_Map table.
- The pages are mapped according to the map data.
- If the initial page information in the map data is null and the page is a read-only page, then it maps the lowest-numbered, currently-mapped page that is in memory for the given vproc. If the page has never been mapped, then page one gets mapped.
- A read/write GLOP or globally modifiable GLOP can only be a single page. It always maps page one.
- Multiple page read-only GLOP data could have different pages mapped to different external routines at the same time.
- When there is no data row for a particular page of a GLOP, but it is in range, a zero filled page is mapped when an external routine pages it in.
- The external routine always has to check to make sure the correct page is mapped for multiple page read-only GLOP data.

## Related Information

For more information on GLOP data, see [Global and Persistent Data](#).

## FNC\_GetGraphicLength

Returns the length, in bytes, for an external routine input argument that has a CHARACTER(*n*) CHARACTER SET GRAPHIC data type.

The length returned is the length of the CHARACTER(n) CHARACTER SET GRAPHIC data when the external routine is invoked.

## Syntax

```
int
FNC_GetGraphicLength(void *inputData);
```

### Syntax Elements

#### *inputData*

a pointer to an external routine input argument that has a data type of CHARACTER(n) CHARACTER SET GRAPHIC.

## Usage Notes

Undefined results occur if the *inputData* argument points to data that is not CHARACTER(n) CHARACTER SET GRAPHIC.

## Example Using FNC\_GetGraphicLength

```
#include <string.h>
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

void strrevcomp_graphic(GRAPHIC *input, VARBYTE *result)
{
    result->length=FNC_GetGraphicLength(input);
    memcpy(&result->bytes[0],input,result->length);
}
```

## FNC\_GetHashAmp

Returns an integer representing the number of the AMP responsible for the specified key.

### Syntax

```
int
FNC_GetHashAmp(FNC_HashRow_t *data,
               int           size,
               int           *retCode)
```

## Syntax Elements

### *data*

IN parameter

A pointer to an array of structures representing table columns.

FNC\_HashRow\_t is defined as follows:

```
typedef struct {
    void *data;
    parm_tx type;
} FNC_HashRow_t;
```

### *size*

IN parameter

The size of the data and return arrays.

### *retCode*

OUT parameter

A pointer to an integer value to indicate success or an error number. 0 indicates success.

## Usage Notes

This routine is callable on a PE vproc only by a table operator.

## FNC\_GetInternalValue

Retrieves the value of a Period, JSON, or DATASET type that is used as an input argument to an external routine.

## Syntax

```
void
FNC_GetInternalValue ( int      typeHandle,
                      void      *returnValue,
                      int      bufSize,
                      int      *length )
```



## Syntax Elements

### *typeHandle*

The handle to a Period, JSON, or DATASET type instance that is defined to be an input parameter to a UDF or UDM, or an IN or INOUT parameter to an external stored procedure.

### *returnValue*

A pointer to a buffer allocated by the user where the return value will be stored.

### *bufSize*

The size in bytes of the *returnValue* buffer.

### *length*

The size in bytes of the return value.

## Usage Notes

### JSON Data Type

FNC\_GetInternalValue retrieves both character and binary JSON values. For text-based values, the routine returns the string representation of the JSON instance, either as UNICODE or LATIN text, depending on how the instance was defined.

Before calling FNC\_GetInternalValue, the external routine must allocate the buffer pointed to by *returnValue*. You can call FNC\_GetJSONInfo or FNC\_GetExtendedJSONInfo to determine the appropriate buffer size, or you can allocate a buffer with the maximum possible length of 64000 bytes.

Use FNC\_GetInternalValue only when the JSON data is *not* stored as a LOB. If FNC\_GetJSONInfo or FNC\_GetExtendedJSONInfo returns numLobs = 0, you can use FNC\_GetInternalValue; otherwise, you should use FNC\_GetJSONInputLob instead.

### DATASET Data Type

Before calling FNC\_GetInternalValue, the external routine must allocate the buffer pointed to by *returnValue*. You can call FNC\_GetDatasetInfo to determine the appropriate buffer size.

Use FNC\_GetInternalValue only when the DATASET data is *not* stored as a LOB. If FNC\_GetDatasetInfo returns dataLob = 0, you can use FNC\_GetInternalValue; otherwise, you should use FNC\_GetDatasetInputLob instead.

For Avro instances, this routine allows a user to access a buffer that contains the UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value. This is equivalent to the transform format.

When retrieving a CSV value, the value will not include any optional schema.

## Period Types

For Period types, the size of the buffer depends on the element type of the Period data type.

IF the period type is ...	THEN the buffer must be large enough to hold ...
PERIOD(DATE)	two DATE values.
PERIOD(TIME)	two TIME values.
PERIOD(TIME WITH TIME ZONE)	two TIME WITH TIME ZONE values.
PERIOD(TIMESTAMP)	two TIMESTAMP values.
PERIOD(TIMESTAMP WITH TIME ZONE)	two TIMESTAMP WITH TIME ZONE values.

To guarantee that the value you pass in for the *bufSize* argument matches the length of the data type, use the following macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example of Getting the Value of a Period Type

```
void check_duration( PDT_HANDLE    *duration,
                    INTEGER        *result,
                    char            sqlstate[6])
{
    void * tmpBuf = 0;
    int bufSize = 0;
    int length;

    /* Get the value of the PERIOD(DATE) duration. */
    bufSize = SIZEOF_DATE * 2;
    tmpBuf = FNC_malloc(bufSize);
    FNC_GetInternalValue(*duration, tmpBuf, bufSize, &length);
}
```

```
...
}
```

## Example of Getting the Value of a JSON Type

This example uses `FNC_GetInternalValue` to retrieve the string representation of a JSON instance, and then search it for a particular name-value pair.

SQL definition:

```
REPLACE FUNCTION getJSONValue (a1 JSON(100))
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getJSONValue!getJSONValue.c!F!getJSONValue';
```

C function definition, `getJSONValue.c`

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
#define buffer_size 64000

void getJSONValue (JSON_HANDLE *json_instance,
                  VARCHAR_LATIN *result,
                  char sqlstate[6])
{
    char *jsonBuf;
    char *searchString1, *searchString2;
    int maxLength = 0;
    charset_et charset = 0;
    int actualLength = 0;
    int strLength = 0;
    int numLobs = 0;

    /* Get the length of the JSON instance. */
    FNC_GetJSONInfo((*json_instance), &maxLength, &charset, &numLobs);

    if (numLobs == 0)
```

```

{
    jsonBuf = (char*)FNC_malloc(maxLength);
    FNC_GetInternalValue((*json_instance),
                        (void*)jsonBuf,
                        maxLength,
                        &actualLength);

    searchString1 = strstr(jsonBuf, "name");
    searchString2 = strchr(searchString1, ':');
    searchString1 = strchr(searchString2, '}');
    strLength = searchString1 - searchString2;
    strncpy(result,(searchString2+sizeof(char)), strLength-sizeof(char));

    FNC_free(jsonBuf);
}
}

```

Example query:

```
SELECT getJSONValue(NEW JSON('{"name":"Cameron"}'));
```

Result:

```

getJSONValue( NEW JSON('{"name":"Cameron"}', LATIN))
-----
"Cameron"

```

## FNC\_GetJSONInfo

Retrieves the maximum length and character set of a JSON type instance that is used as an input argument to an external routine. The function also indicates whether or not the JSON data is stored in a LOB subtable.

### Syntax

```

void
FNC_GetJSONInfo ( JSON_HANDLE  jsonHandle,
                  int          *maxLength,
                  charset_et   *charset,
                  int          *numLobs)

```

## Syntax Elements

### *jsonHandle*

a handle to a JSON type instance.

JSON\_HANDLE is defined in sqltypes\_td.h as:

```
typedef int JSON_HANDLE;
```

### *maxLength*

the maximum possible length of the JSON instance in bytes.

### *charset*

the character set of the JSON text, which is either UNICODE or LATIN.

### *numLobs*

the number of LOBs used to store the JSON data. Valid values are as follows:

- 0, which indicates that the data is *not* stored as a LOB
- 1, which indicates that the data is stored as a LOB

## Usage Notes

FNC\_GetJSONInfo takes a JSON type instance as an input argument. The JSON type instance is passed to the function as a JSON handle. The function returns:

- The maximum possible length, in bytes, of the JSON type instance specified by *jsonHandle*. This information is returned in the *maxLength* parameter.
- The character set of the JSON type instance specified by *jsonHandle*. This information is returned in the *charset* parameter. The return value is one of the following as defined in sqltypes\_td.h:
  - LATIN\_CT
  - UNICODE\_CT
- An indication of whether or not the JSON data is stored as a LOB. This is specified by the *numLobs* parameter.

Based on the *numLobs* value, the user can determine which FNC routines to use for reading and writing JSON data:

- If the JSON data is *not* stored as a LOB, you must use FNC\_GetInternalValue or FNC\_SetInternalValue.
- If the JSON data is stored as a LOB, you must use FNC\_GetJSONInputLob or FNC\_GetJSONResultLob together with the LOB FNC routines.

## Example Using FNC\_GetJSONInfo

SQL definition:

```
REPLACE FUNCTION getJSONInfo (a1 TD_ANYTYPE)
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getJSONInfo!getJSONInfo.c!F!getJSONInfo';
```

C function definition, getJSONInfo.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
#define buffer_size 64000

void getJSONInfo( JSON_HANDLE *json_instance,
                  VARCHAR_LATIN *result,
                  char sqlstate[6])
{
    int length = 0;
    charset_et charset = 0;
    int numLobs = 0;

    /* Get the info of the JSON instance. */
    FNC_GetJSONInfo((*json_instance), &length, &charset, &numLobs);
    sprintf(result, "Length: %d, CharSet: %d, NumLobs: %d\0", length,
    charset, numLobs);
}
```

Example table, data, and query:

```
CREATE TABLE jsonTable(id INTEGER, j JSON(100) CHARACTER SET LATIN);

INSERT INTO jsonTable(1, '{"name":"Cameron"}');

SELECT getJSONInfo(j) FROM jsonTable;
```

Result:

```
getJSONInfo(j)
-----
Length: 100, CharSet: 1, NumLobs: 0
```

## FNC\_GetJSONInputLob

Returns a LOB\_LOCATOR for a JSON instance which has its data stored as a LOB. You can use this LOB\_LOCATOR with LOB FNC routines to read the data from the JSON instance.

### Syntax

```
void
FNC_GetJSONInputLob ( JSON_HANDLE   jsonHandle,
                      LOB_LOCATOR   *object )
```

### Syntax Elements

#### *jsonHandle*

A handle to a JSON type instance that is defined to be an input argument to an external routine.

JSON\_HANDLE is defined in sqltypes\_td.h as typedef int JSON\_HANDLE;

#### *object*

A pointer to a LOB\_LOCATOR that you can use with the LOB FNC routines to read the LOB data of a JSON instance.

### Usage Notes

JSON data for any particular instance may be stored in the base table row or in a LOB subtable. Use FNC\_GetJSONInputLob only when the JSON data is stored as a LOB. If FNC\_GetJSONInfo or FNC\_GetExtendedJSONInfo returns numLobs > 0, you can use FNC\_GetJSONInputLob; otherwise, you should use FNC\_GetInternalValue instead.

You can use FNC\_GetJSONInputLob to retrieve both character and binary JSON data.

### Example Using FNC\_GetJSONInputLob

This example uses FNC\_GetJSONInputLob to retrieve the string representation of a JSON instance, and then search it for a particular name-value pair.

SQL definition:

```
REPLACE FUNCTION getJSONInput (a1 JSON(100000))
RETURNS VARCHAR(100)
```

```

NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getJSONInput!getJSONInput.c!F!getJSONInput';

```

C function definition, getJSONInput.c

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
#define buffer_size 64000
void getJSONInput (JSON_HANDLE *json_instance,
                   VARCHAR_LATIN *result,
                   char sqlstate[6])
{
    char *jsonBuf;
    char *searchString1, *searchString2;
    int maxLength = 0;
    charset_et charset = 0;
    int actualLength = 0;
    int strLength = 0;
    int numLobs = 0;
    /* Get the length of the JSON instance. */
    FNC_GetJSONInfo((*json_instance), &maxLength, &charset, &numLobs);
    if (numLobs != 0)
    {
        LOB_LOCATOR inLOB;
        LOB_CONTEXT_ID id;
        FNC_LobLength_t readlen, actualInputLength;
        int trunc_err = 0;
        BYTE *bufPtr = 0;
        jsonBuf = (char*)FNC_malloc(maxLength);
        FNC_GetJSONInputLob(*json_instance, &inLOB);
        readlen=0;
        actualInputLength = 0;
        bufPtr = jsonBuf;
        FNC_LobOpen(inLOB, &id, 0, 0);
        while( FNC_LobRead(id, bufPtr, buffer_size, &readlen) == 0 && !trunc_err )
        {
            bufPtr += readlen;
            actualInputLength += readlen;
            if (actualInputLength >= maxLength)

```



```

    {
        trunc_err = 1;
        actualInputLength = maxLength;
    }
    /* check trunc_err and properly report an error
       (performed in the same way as for a standard UDF) */
}
FNC_LobClose(id);
searchString1 = strstr(jsonBuf, "name");
searchString2 = strchr(searchString1, ':');
searchString1 = strchr(searchString2, '}');
strLength = searchString1 - searchString2;
strncpy(result, (searchString2+sizeof(char)), strLength-sizeof(char));
FNC_free(jsonBuf);
}
}

```

Example table, data, and query:

```

CREATE TABLE jsonTable(id INTEGER, j JSON(100000));
INSERT INTO jsonTable(1, <data large enough to be stored as LOB>);
SELECT getJSONInput(j) FROM jsonTable;

```

Result:

```
"Cameron"
```

## FNC\_GetJSONResultLob

Returns a LOB\_RESULT\_LOCATOR to a LOB associated with a JSON instance. You can use this LOB\_RESULT\_LOCATOR with LOB FNC routines to write data to the JSON instance.

## Syntax

```

void
FNC_GetJSONResultLob ( JSON_HANDLE      jsonHandle,
                      LOB_RESULT_LOCATOR *object )

```

### Syntax Elements

#### *jsonHandle*

a handle to a JSON type instance that is defined to be a result parameter to an external routine.

JSON\_HANDLE is defined in sqltypes\_td.h as:

```
typedef int JSON_HANDLE;
```

### **object**

a pointer to a LOB\_RESULT\_LOCATOR that you can use with the LOB FNC routines to write the LOB data of a JSON instance.

## **Usage Notes**

JSON data for any particular instance may be stored in the base table row or in a LOB subtable. Use FNC\_GetJSONResultLob only when the JSON data will be stored as a LOB. If FNC\_GetJSONInfo or FNC\_GetExtendedJSONInfo returns numLobs > 0, you can use FNC\_GetJSONResultLob; otherwise, you should use FNC\_SetInternalValue instead.

You can use FNC\_GetJSONResultLob to set both character and binary JSON data.

## **Example Using FNC\_GetJSONResultLob**

This example uses FNC\_GetJSONResultLob to create a JSON instance from a CLOB.

SQL definition:

```
REPLACE FUNCTION getJSONResult (a1 CLOB AS LOCATOR)
RETURNS JSON CHARACTER SET LATIN
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!getJSONResult!getJSONResult.c!F!getJSONResult';
```

C function definition, getJSONResult.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
#define buffer_size 64000

void getJSONResult (LOB_LOCATOR *inLOB,
                   JSON_HANDLE *json_instance,
                   char sqlstate[6])
{
    LOB_RESULT_LOCATOR outLOB;
```

```

BYTE buffer[buffer_size];
LOB_CONTEXT_ID id;
FNC_LobLength_t readlen, writelen, actualInputLength;
int trunc_err = 0;
int length = 0;
charset_et charset = 0;
int numLobs = 0;

FNC_GetJSONInfo((*json_instance), &length, &charset, &numLobs);

FNC_LobOpen(*inLOB, &id, 0, 0);
actualInputLength = FNC_GetLobLength(*inLOB);
if (actualInputLength > length)
{
    sprintf(sqlstate, "U0101\0");
    return;
}

FNC_GetJSONResultLob(*json_instance,&outLOB);

readlen=0;
writelen=0;

while( FNC_LobRead(id, buffer, buffer_size, &readlen) == 0 && !trunc_err )
{
    trunc_err = FNC_LobAppend(outLOB, buffer, readlen, &writelen);
    /* check trunc_err and properly report an error
       (performed in the same way as for a standard UDF) */
}

FNC_LobClose(id);
sprintf(sqlstate, "00000\0");
}

```

Example table, data, and query:

```

CREATE TABLE clobTable(id INTEGER, c CLOB);

INSERT INTO clobTable(1, '{"CompanyName":"Teradata"}');

SELECT getJSONResult(c).JSONExtractValue('$.CompanyName')

FROM clobTable;

```

Result:

```
getJSONResult(c).JSONEXTRACTVALUE('$$.CompanyName')
-----
Teradata
```

## FNC\_GetLobLength

Returns the length, in bytes, of a BLOB or CLOB.

### Syntax

```
FNC_LobLength_t
FNC_GetLobLength ( LOB_LOCATOR  object )
```

### Syntax Elements

***object***

the object.

### Usage Notes

The LOB\_LOCATOR argument is validated to the extent possible. If the argument is not valid, the request that invoked the UDF, UDM, or external stored procedure fails and typically returns an error that looks like this:

```
7554 Invalid LOCATOR argument to LOB access function in UDF
    database_name.udf_name.
```

Control does not return to the UDF, UDM, or external stored procedure.

### Disk Access

FNC\_GetLobLength does not cause a disk read operation.

### Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_GetLobLength

```
void concat2( LOB_LOCATOR      *a,
              LOB_LOCATOR      *b,
              LOB_RESULT_LOCATOR *result,
              char               sqlstate[6] )
{
    FNC_LobLength_t len;

    len = FNC_GetLobLength(*a);

    ...

}
```

## FNC\_GetLobLength\_CL

Returns the length, in bytes, of a BLOB or CLOB.

### Syntax

```
FNC_LobLength_t
FNC_GetLobLength_CL ( void  *locator )
```

### Syntax Elements

#### *locator*

the client locator.

### Usage Notes

This function is used with table operators and differs from FNC\_GetLobLength in the data type of the input parameter.

## Example Using FNC\_GetLobLength\_CL

```
BYTE          *value;
int            locatorLength;
FNC_LobLength_t lobLength;
FNC_TblOpHandle_t *Handle;
int            i;           // column index
int            null_ind;
```

```

....

// read locator
FNC_TblOpGetAttributeByNdx(Handle, i, (void **) &value,
&null_ind, &locatorLength);
if (null_ind != -1)
    // get LOB length
    lobLength = FNC_GetLobLength_CL(value);

```

## FNC\_GetOutputBufferSize

Returns the size of the output buffer that the scalar or aggregate UDF must return as the result, an integer value.

### Syntax

```

int
FNC_GetOutputBufferSize();

```

### Usage Notes

A UDF that is used for the algorithmic decompression of previously compressed character columns must call `FNC_GetOutputBufferSize` to determine the maximum length of the string to build for the return argument. This is necessary because the UDF can declare a `BYTE(n)`, `VARBYTE(n)`, `CHAR(n)`, or `VARCHAR(n)` return argument where the value of *n* is greater than or equal to the size of *n* for the column data type.

This function can only be called from within a scalar or aggregate UDF.

### Restrictions

This function can only be called from within a scalar or aggregate UDF.

## FNC\_GetPhase

Returns information that a table function uses to determine how it was called in the FROM TABLE clause of the SELECT statement and what action to perform.

Returns an `FNC_Mode` value that indicates whether the arguments to the table function are constant or vary.

`FNC_Mode` is defined as:

```

typedef enum FNC_Mode
{

```

```
TBL_MODE_VARY = 1,
TBL_MODE_CONST = 2
} FNC_Mode;
```

This value ...	Means that the table function arguments ...
TBL_MODE_CONST	are constant.
TBL_MODE_VARY	vary and are based on the rows produced by the correlated table specification in the SELECT statement. The table function might only be called on specific AMP vprocs in this mode.

## Syntax

```
FNC_Mode
FNC_GetPhase(FNC_Phase *Phase)
```

### Syntax Elements

#### *Phase*

the processing phase in which the function was called and the action that the table function should perform.

FNC\_Phase is defined as:

```
typedef enum FNC_Phase {
    TBL_PRE_INIT=20,
    TBL_INIT=21,
    TBL_BUILD=22,
    TBL_FINI=23,
    TBL_END=24,
    TBL_ABORT=25,
    TBL_BUILD_EOF=26
} FNC_Phase;
```

For the meanings of FNC\_Phase values, see [Processing Phase](#).

## Usage Notes

A table function calls FNC\_GetPhase before doing anything else.

For a table function that is passed in a variable set of input arguments (where the mode is TBL\_MODE\_VARY) and needs to know when it is being passed in the last logical, or qualified, row on an AMP, the best practice is to use FNC\_GetPhaseEx instead of FNC\_GetPhase. For details, see [FNC\\_GetPhaseEx](#).

## Processing Phase

The processing phases that FNC\_GetPhase can return depend on the mode.

IF the value of FNC_Mode is ...	THEN this value for Phase ...	Means that the table function ...
TBL_MODE_VARY	TBL_PRE_INIT	is being called for the first time for all the rows that it will be called for. The input arguments to the function contain the first set of data. During this phase, the function has an opportunity to establish overall global context, but should not build any result row. The function continues to the TBL_INIT phase.
	TBL_INIT	should open any connections to external objects, such as files, if there is a need to do so. The input arguments to the function contain the first set of data. During this phase, the function should not build any result row. The function continues to the TBL_BUILD phase.
	TBL_BUILD	should fill out the result arguments to build a row. The function remains in the TBL_BUILD phase until it sets the sqlstate argument to "02000" to indicate no data, whereupon it continues to the TBL_FINI phase.
	TBL_FINI	should close any connections, such as file handles, that were opened during the TBL_INIT phase. If there is more variable input data, the function returns to the TBL_INIT phase. Otherwise, the function continues to the TBL_END phase.
	TBL_END	should close all external connections and release any scratch memory it might have allocated. The table function is not called again after this phase.
	TBL_ABORT	is being aborted and should close all external connections and release any previously-allocated memory. A function can be called at any time with this phase, which is only entered when one of the table functions calls the library function FNC_TblAbort. It is not entered when the function is aborted for an external reason, such as a user abort.
TBL_MODE_CONST	TBL_PRE_INIT	may decide whether it should be the controlling copy of all table functions running on other AMP vprocs. If the function wants to provide control context to all other copies of the table function, the function must call FNC_TblControl. If the function does not want to be the controlling copy of the table function, or if the function is designed without the need for a controlling function, the function can simply return and do nothing during this phase. All copies of the table function must complete this phase before any copy continues to the TBL_INIT phase.
	TBL_INIT	should open any connections to external objects, such as files, if there is a need to do so.



IF the value of FNC_Mode is ...	THEN this value for Phase ...	Means that the table function ...
		Any copy of the function that does not want to participate further must call FNC_TblOptOut. After the function returns, it will not be called again. All copies of the table function must complete this phase before any copy continues to the TBL_BUILD phase.
	TBL_BUILD	should fill out the result arguments to build a row. The function remains in the TBL_BUILD phase until it sets the sqlstate argument to "02000" to indicate no data, whereupon it continues to the TBL_END phase.
	TBL_END	should close all external connections and release any scratch memory it might have allocated. The table function is not called again after it returns from this phase. The controlling copy of the table function, if one exists, is called with this phase after all other copies of the table function have completed this phase, which allows the controlling function to do any final cleanup or notification to the external world.
	TBL_ABORT	is being aborted and should close all external connections and release any previously-allocated memory. A function can be called at any time with this phase, which is only entered when one of the table functions calls the library function FNC_TblAbort. It is not entered when the function is aborted for an external reason, such as a user abort.

## Restrictions

This function can only be called from within a table function. Calling this function from an external stored procedure, scalar UDM, scalar function, or aggregate function results in an exception on the transaction.

A table function can call FNC\_GetPhase or FNC\_GetPhaseEx, but cannot call both.

## Example Using FNC\_GetPhase

```

FNC_Phase    Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            ...
            break;
        }
    }
}

```

```

    case TBL_INIT:
    {
        ...
        break;
    }
    case TBL_BUILD:
    {
        ...
        break;
    }
    ...
}
...

```

## FNC\_GetPhaseEx

An extended version of FNC\_GetPhase that provides additional options for variable mode table functions.

FNC\_GetPhaseEx returns an FNC\_Mode value that indicates whether the arguments to the table function are constant or vary.

FNC\_Mode is defined as:

```

typedef enum FNC_Mode
{
    TBL_MODE_VARY = 1,
    TBL_MODE_CONST = 2
} FNC_Mode;

```

This value ...	Means that the table function arguments ...
TBL_MODE_CONST	are constant.
TBL_MODE_VARY	vary and are based on the rows produced by the correlated table specification in the SELECT statement. The table function might only be called on specific AMP vprocs in this mode.

## Syntax

```

FNC_Mode
FNC_GetPhaseEx(FNC_Phase *Phase,
               int          option)

```

## Syntax Elements

### Phase

The processing phase in which the function was called and the action that the table function should perform.

FNC\_Phase is defined as:

```
typedef enum FNC_Phase {
    TBL_PRE_INIT=20,
    TBL_INIT=21,
    TBL_BUILD=22,
    TBL_FINI=23,
    TBL_END=24,
    TBL_ABORT=25,
    TBL_BUILD_EOF=26
} FNC_Phase;
```

For the meanings of FNC\_Phase values, see [Processing Phase](#).

### option

A value of 0-5.

The `sqltypes_td.h` header file provides the following constants that you can use:

```
#define TBL_NOOPTIONS    0
#define TBL_LASTROW     1
#define TBL_NEWROW      2
#define TBL_NEWROWEOF   4
```

A value of 3 specifies both `TBL_NEWROW` and `TBL_LASTROW` options.

A value of 5 specifies both `TBL_NEWROWEOF` and `TBL_LASTROW` options.

The following options are valid only for variable mode table functions. These options are ignored if you specify them in a constant mode table function.

Options	Value	Description and Usage
TBL_NOOPTIONS	0	Indicates that no options are specified. Use this option when you only want to retrieve the processing phase in which the function was called.
TBL_LASTROW	1	Allows a function to determine when it is being passed the last input row on an AMP. Use this option if your table function processes a set of input rows before returning an output row. You must set the EOF indicator when using TBL_LASTROW to signal the end of the processing when the last row has been encountered. The

Options	Value	Description and Usage
		function then moves from the TBL_BUILD phase to the TBL_BUILD_EOF phase where the row is built.
TBL_NEWROW	2	If this option is set and the phase is TBL_BUILD, the function is called with a new row with a phase of TBL_BUILD. Use this option when you want to get a new row on each function invocation. If end of file, TBL_LASTROW or ProcessLastRow is true, then this option is ignored.
TBL_NEWROW and TBL_LASTROW	3	The behavior for both the TBL_NEWROW and TBL_LASTROW options are in effect.
TBL_NEWROWEOF	4	If this option is set, and the phase is TBL_BUILD and EOF is set, then the function is called with a new row with a phase of TBL_BUILD. Use this option when you want to get a new row when EOF is signaled. If end of file, TBL_LASTROW or ProcessLastRow is true, then this option is ignored.
TBL_NEWROWEOF and TBL_LASTROW	5	The behavior for both the TBL_NEWROWEOF and TBL_LASTROW options are in effect.

For example, calling FNC\_GetPhaseEx with the option TBL\_NOOPTIONS does not reset the TBL\_NEWROW or TBL\_NEWROWEOF behavior.

If you specify TBL\_LASTROW, the option remains in effect for the duration of the request. The TBL\_NEWROW and TBL\_NEWROWEOF options are set and reset based on the options specified during each call to FNC\_GetPhaseEx where the option value is nonzero.

## Usage Notes

A table function calls FNC\_GetPhaseEx before doing anything else.

Although FNC\_GetPhaseEx can be used successfully by a table function that is passed in constant arguments (where the return value of FNC\_GetPhaseEx is TBL\_MODE\_CONST), it is most useful to a table function that is passed in variable arguments (where the return value of FNC\_GetPhaseEx is TBL\_MODE\_VARY).

The functionality that FNC\_GetPhaseEx provides complements the HASH BY and LOCAL ORDER BY clauses that order input to table functions. For an example on how to order input arguments to a table function, see [Ordering Input Arguments to Table UDFs](#).

FNC\_GetPhaseEx also provides users with more control of table phase transitions when developing table functions. Users can reduce the number of phase transitions required during execution of a table function, thus reducing the number of UDF invocations and improving table function performance.

## Processing Phase

FNC\_GetPhaseEx returns the current phase of the table function in the *phase* argument. The processing phases that FNC\_GetPhaseEx can return depend on the mode.

IF the value of FNC_Mode is ...	THEN this value for Phase ...	Means that the table function ...
TBL_MODE_VARY	TBL_PRE_INIT	is being called for the first time for all the rows that it will be called for. The input arguments to the function contain the first set of data. During this phase, the function has an opportunity to establish overall global context, but should not build any result row. The function continues to the TBL_INIT phase.
	TBL_INIT	should open any connections to external objects, such as files, if there is a need to do so. The input arguments to the function contain the first set of data. During this phase, the function should not build any result row. The function continues to the TBL_BUILD phase.
	TBL_BUILD	should continue to process input arguments. If the TBL_NEWROW option is set, then call the table function with a new row with a phase of TBL_BUILD. If the TBL_NEWROWEOF option and EOF are set, then call the table function with a new row with a phase of TBL_BUILD. If the TBL_LASTROW option is set, the function remains in the TBL_BUILD phase until it is passed in the last set of data, where it continues to the TBL_BUILD_EOF phase.
	TBL_BUILD_EOF	should fill out the result arguments to build a row. The function remains in the TBL_BUILD_EOF phase until it sets the sqlstate argument to "02000" to indicate no data, whereupon it continues to the TBL_END phase.
	TBL_FINI	should close any connections, such as file handles, that were opened during the TBL_INIT phase. If there is more variable input data, the function returns to the TBL_INIT phase. Otherwise, the function continues to the TBL_END phase.
	TBL_END	should close all external connections and release any scratch memory it might have allocated. The table function is not called again after this phase.
	TBL_ABORT	is being aborted and should close all external connections and release any previously-allocated memory. A function can be called at any time with this phase, which is only entered when one of the table functions calls the library function FNC_TblAbort. It is not entered when the function is aborted for an external reason, such as a user abort.

IF the value of FNC_Mode is ...	THEN this value for Phase ...	Means that the table function ...
TBL_MODE_CONST	TBL_PRE_INIT	<p>may decide whether it should be the controlling copy of all table functions running on other AMP vprocs.</p> <p>If the function wants to provide control context to all other copies of the table function, the function must call FNC_TblControl.</p> <p>If the function does not want to be the controlling copy of the table function, or if the function is designed without the need for a controlling function, the function can simply return and do nothing during this phase.</p> <p>All copies of the table function must complete this phase before any copy continues to the TBL_INIT phase.</p>
	TBL_INIT	<p>should open any connections to external objects, such as files, if there is a need to do so.</p> <p>Any copy of the function that does not want to participate further must call FNC_TblOptOut. After the function returns, it will not be called again.</p> <p>All copies of the table function must complete this phase before any copy continues to the TBL_BUILD phase.</p>
	TBL_BUILD	<p>should fill out the result arguments to build a row.</p> <p>The function remains in the TBL_BUILD phase until it sets the sqlstate argument to "02000" to indicate no data, whereupon it continues to the TBL_END phase.</p>
	TBL_END	<p>should close all external connections and release any scratch memory it might have allocated. The table function is not called again after it returns from this phase.</p> <p>The controlling copy of the table function, if one exists, is called with this phase after all other copies of the table function have completed this phase, which allows the controlling function to do any final cleanup or notification to the external world.</p>
	TBL_ABORT	<p>is being aborted and should close all external connections and release any previously-allocated memory. A function can be called at any time with this phase, which is only entered when one of the table functions calls the library function FNC_TblAbort. It is not entered when the function is aborted for an external reason, such as a user abort.</p>

## Restrictions

This function can only be called from within a table function. Calling this function from an external stored procedure, scalar UDM, scalar function, or aggregate function results in an exception on the transaction.

A table function can use either FNC\_GetPhase or FNC\_GetPhaseEx, but cannot use both.

## Example Using FNC\_GetPhaseEx

```

FNC_Phase    Phase;

if (FNC_GetPhaseEx(&Phase, TBL_LASTROW) == TBL_MODE_VARY)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            ...
            break;
        }
        case TBL_INIT:
        {
            ...
            break;
        }
        case TBL_BUILD:
        {
            ...
            break;
        }
        case TBL_BUILD_EOF:
        {
            ...
            break;
        }
        ...
    }
}
...

```

## Example Using FNC\_GetPhaseEx

By using the FNC\_GetPhaseEx options, you can reduce the number of table phase transitions and improve table function performance in the following situations:

- In a 1:1 (one row in : one row out) processing mode, use the TBL\_NEWROW option to get a new row on every TBL\_BUILD call.

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_NEWROW);
```

- In a 1:M (one row in : many rows out) processing mode, use the TBL\_NEWROWEOF option to get a new row when EOF is signaled.

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_NEWROWEOF);
```

- In a M:1 (many rows in : one row out) processing mode, you can use:

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_LASTROW | TBL_NEWROW);
```

In a combined M:1 and 1:M processing mode, you can use the following that does not pass a new row until EOF:

```
FNC_Mode mode = FNC_GetPhaseEx(&thePhase, TBL_LASTROW | TBL_NEWROWEOF);
```

## FNC\_GetQueryBand

Returns the concatenated query band string of the current query bands for the session (transaction, session, profile query bands) in the character set of the function (LATIN or UNICODE).

### Syntax

```
void
FNC_GetQueryBand ( void *QBandBuf,
                  int   BufSize,
                  int   *QBandLen )
```

### Syntax Elements

#### *QBandBuf*

a pointer to a buffer that FNC\_GetQueryBand (or FNC\_GetQueryBandU) uses to return the query band. Before calling FNC\_GetQueryBand (or FNC\_GetQueryBandU), you must allocate the buffer pointed to by *QBandBuf*. The buffer must be large enough to return the query band plus a null terminator.

If the query band contains name-value pairs for the transaction, session, and/or profile, the function returns the concatenated transaction, session, and/or profile query band text.

For example, if the query band contains name-value pairs for the transaction, session, and profile, the function returns the concatenated transaction, session, and profile query band text as follows:

```
=T> transaction_query_band  =S> session_query_band
=P> profile_query_band
```



Similarly, if the query band contains name-value pairs for the transaction and session, the function returns the concatenated transaction and session query band text as follows:

```
=T> transaction_query_band =S> session_query_band
```

If the query band contains name-value pairs for the transaction only, the text contains:

```
=T> transaction_query_band
```

If the query band contains name-value pairs for the session only, the text contains:

```
=S> session_query_band
```

If the query band contains name-value pairs for the profile only, the text contains:

```
=P> profile_query_band
```

If there are no name-value pairs for the transaction, session, or profile, the return string is 0 bytes.

### **BufSize**

the size in bytes of the *QBandBuf* buffer.

The `sqltypes_td.h` header file provides the following constants that you can use:

```
#define FNC_MAXQUERYBANDSIZE 12304
#define FNC_MAXQUERYBANDSIZE_U 24608
```

where `FNC_MAXQUERYBANDSIZE` is the maximum query band size in bytes and `FNC_MAXQUERYBANDSIZE_U` is the maximum Unicode query band size in bytes.

### **QBandLen**

the size in bytes of the returned query band. The length includes the size of any null termination characters.

## **Usage Notes**

The character set of a UDF is determined by the character set of the user that creates the function. Therefore, if the user creating a UDF has a character set of LATIN, the UDF has a character set of LATIN. If the user creating a UDF has a character set of UNICODE, the UDF has a character set of UNICODE. `FNC_GetQueryBand` assumes the input character parameters are in the function character set and returns the output parameters in the same character set.

This interface can be called by a UDF, UDM, or external stored procedure to retrieve the current query band, a string of name-value pairs that can be set on a session, transaction, or profile to identify the originating source of queries and help manage task priorities and track system use.

IF you want to ...	THEN use the ...
set the query band for a session or transaction	SET QUERY_BAND SQL statement. For details, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
add or modify a query band for a profile	<ul style="list-style-type: none"> <li>• CREATE PROFILE</li> <li>• MODIFY PROFILE</li> </ul> For details, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
get transaction, session, or profile name-value pairs from the query band string that FNC_GetQueryBand returns	FNC_GetQueryBandPairs function. For details, see <a href="#">FNC_GetQueryBandPairs</a> .
search the transaction, session, or profile name-value pairs in the query band string that FNC_GetQueryBand returns and get the value that corresponds to a specific name	FNC_GetQueryBandValue function. For details, see <a href="#">FNC_GetQueryBandValue</a> .

## System Routines that Use FNC\_GetQueryBand

Teradata provides a UDF called GetQueryBand and an external stored procedure called GetQueryBandSP that retrieve the query band using the FNC\_GetQueryBand library function. GetQueryBand and GetQueryBandSP are created in the SYSLIB database when you execute the DIPDEM script using the DIP utility.

For more information on DIP, see *Teradata Vantage™ - Database Utilities*, B035-1102. For more information on GetQueryBand and GetQueryBandSP, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_GetQueryBand

```
#define SQL_TEXT Latin_Text
#include<sqltypes_td.h>
#include<string.h>

void xsp_getqryband ( SQL_TEXT *QBandBuf, char sqlstate[6] )
{
    int QBandLen;
```

```

    FNC_GetQueryBand(QBandBuf, FNC_MAXQUERYBANDSIZE, &QBandLen);

    strcpy(sqlstate, "00000"); // returns exception type

    return ;
}

```

## FNC\_GetQueryBandU

Returns the concatenated query band string of the current query bands for the session (transaction, session, profile query bands) in character set UNICODE.

### Syntax

```

void
FNC_GetQueryBandU ( void   *QBandBuf,
                    int     BufSize,
                    int     *QBandLen )

```

### Syntax Elements

#### *QBandBuf*

a pointer to a buffer that FNC\_GetQueryBand (or FNC\_GetQueryBandU) uses to return the query band. Before calling FNC\_GetQueryBand (or FNC\_GetQueryBandU), you must allocate the buffer pointed to by *QBandBuf*. The buffer must be large enough to return the query band plus a null terminator.

If the query band contains name-value pairs for the transaction, session, and/or profile, the function returns the concatenated transaction, session, and/or profile query band text.

For example, if the query band contains name-value pairs for the transaction, session, and profile, the function returns the concatenated transaction, session, and profile query band text as follows:

```

=T>  transaction_query_band  =S>  session_query_band
=P>  profile_query_band

```

Similarly, if the query band contains name-value pairs for the transaction and session, the function returns the concatenated transaction and session query band text as follows:

```

=T>  transaction_query_band  =S>  session_query_band

```

If the query band contains name-value pairs for the transaction only, the text contains:

```
=T> transaction_query_band
```

If the query band contains name-value pairs for the session only, the text contains:

```
=S> session_query_band
```

If the query band contains name-value pairs for the profile only, the text contains:

```
=P> profile_query_band
```

If there are no name-value pairs for the transaction, session, or profile, the return string is 0 bytes.

### **BufSize**

the size in bytes of the *QBandBuf* buffer.

The `sqltypes_td.h` header file provides the following constants that you can use:

```
#define FNC_MAXQUERYBANDSIZE 12304
#define FNC_MAXQUERYBANDSIZE_U 24608
```

where `FNC_MAXQUERYBANDSIZE` is the maximum query band size in bytes and `FNC_MAXQUERYBANDSIZE_U` is the maximum Unicode query band size in bytes.

### **QBandLen**

the size in bytes of the returned query band. The length includes the size of any null termination characters.

## **Usage Notes**

This interface can be called by a UDF, UDM, or external stored procedure to retrieve the current query band, a string of name-value pairs that can be set on a session, transaction, or profile to identify the originating source of queries and help manage task priorities and track system use.

IF you want to ...	THEN use the ...
set the query band for a session or transaction	SET QUERY_BAND SQL statement. For details, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
add or modify a query band for a profile	<ul style="list-style-type: none"> <li>• CREATE PROFILE</li> <li>• MODIFY PROFILE</li> </ul> For details, see <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

IF you want to ...	THEN use the ...
get transaction, session, or profile name-value pairs from the query band string that FNC_GetQueryBandU returns	FNC_GetQueryBandPairsU function. For details, see <a href="#">FNC_GetQueryBandPairsU</a> .
search the transaction, session, or profile name-value pairs in the query band string that FNC_GetQueryBandU returns and get the value that corresponds to a specific name	FNC_GetQueryBandValueU function. For details, see <a href="#">FNC_GetQueryBandValueU</a> .

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_GetQueryBandU

### SQL Definition

```

REPLACE FUNCTION GetQueryBandExample()
  RETURNS VARCHAR(6159) CHARACTER SET UNICODE
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'CS!GetQBExample!GetQBExample.c';

```

### C Function Definition

```

#define SQL_TEXT Latin_Text

#include<string.h>
#include<sqltypes_td.h>

extern void
FNC_GetQueryBandU(void      *QBandBuf,
                  int        BufSize,
                  int        *QBandLen);

/*****

void GetQueryBandExample (
    VARCHAR_UNICODE *QBandBuf,

```

```

        char sqlstate[6])
{
    int QBandLen;
    int MaxQBLen;

    MaxQBLen = FNC_MAXQUERYBANDSIZE_U;

    FNC_GetQueryBandU(QBandBuf, MaxQBLen, &QBandLen);

    return;
}

```

## FNC\_GetQueryBandPairs

Retrieves transaction, session, or profile name-value pairs from the query band string pointed to by the *QBandBuf* input argument. The query band names and values will be in the character set of the function (LATIN or UNICODE).

FNC\_Get\_QueryBandPairs returns a pointer to an FNC\_QB\_Pair\_t structure.

FNC\_QB\_Pair\_t is defined in `sqltypes_td.h` and has the following members:

Member ...	Specifies ...
QBName	the query band name.
QBValue	the query band value.

The character set of a UDF is determined by the character set of the user that creates the function. Therefore, if the user creating a UDF has a character set of LATIN, the UDF has a character set of LATIN. If the user creating a UDF has a character set of UNICODE, the UDF has a character set of UNICODE. FNC\_GetQueryBandPairs assumes the input character parameters are in the function character set and returns the output parameters in the same character set.

FNC\_GetQueryBandPairs allocates local storage for the return structure. If the returned number of pairs is greater than zero, the UDF, UDM, or external stored procedure must call FNC\_free to free the return structure before exiting.

## Syntax

```

FNC_QB_Pair_t *
FNC_GetQueryBandPairs ( void          *QBandBuf,
                        FNC_QBSearch_et SearchType,
                        int            *NumPairs )

```

## Syntax Elements

### ***QBandBuf***

a pointer to a buffer containing the query band, where the query band can be:

- Returned by FNC\_GetQueryBand
- Retrieved from the DBC.DBQLogTbl.QueryBand column
- Specified by the caller

### ***SearchType***

whether FNC\_GetQueryBandPairs searches for name-value pairs in the transaction, session, and/or profile query band.

FNC\_QBSearch\_et is defined in sqltypes\_td.h and includes the following values:

- QB\_FIRST specifies to return the first unique name-value pair for each name found in the transaction, session, and profile query bands. If the transaction, session, and profile query bands contain the same name, FNC\_GetQueryBandPairs returns the first name-value pair found in the query band in the following order:
  - Transaction query band
  - Session query band
  - Profile query band
- QB\_TXN specifies to return name-value pairs in the transaction query band.
- QB\_SESSION specifies to return name-value pairs in the session query band.
- QB\_PROFILE specifies to return name-value pairs in the profile query band.

### ***NumPairs***

the number of name-value pairs that FNC\_GetQueryBandPairs returns.

## Usage Notes

### System Function that Uses FNC\_GetQueryBandPairs

Teradata provides a table function called GetQueryBandPairs that retrieves name-value pairs in the query band using the FNC\_GetQueryBandPairs library function. GetQueryBandPairs is created in the SYSLIB database when you execute the DIPDEM script using the DIP utility.

For more information on DIP, see *Teradata Vantage™ - Database Utilities*, B035-1102. For more information on GetQueryBandPairs, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

## Restrictions

An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

## Example Using FNC\_GetQueryBandPairs

```
#define SQL_TEXT Latin_Text
#include<sqltypes_td.h>
#include<string.h>

void getPairs ( int *NumPairs, char sqlstate[6] )
{
    FNC_QB_Pair_t *pairPtr;
    char          *QBandBuf;
    int           QBLen;

    QBandBuf = FNC_malloc(FNC_MAXQUERYBANDSIZE);
    if (QBandBuf == NULL)
    {
        strcpy(sqlstate, "U0004");
        strcpy((char *) error_message, "malloc failed");
    }
    else
    {
        FNC_GetQueryBand(QBandBuf, FNC_MAXQUERYBANDSIZE, &QBLen);
        if (QBLen > 0)
        {
            pairPtr = FNC_GetQueryBandPairs(QBandBuf, QB_FIRST, NumPairs);
            if (*NumPairs > 0)
                FNC_free(pairPtr);
        }
        strcpy(sqlstate, "00000");
    }
    FNC_free(QBandBuf);
    return;
}
```

## FNC\_GetQueryBandPairsU

Retrieves transaction, session, or profile name-value pairs from the query band string pointed to by the *QBandBuf* input argument. The query band names and values will be in the UNICODE character set.

FNC\_Get\_QueryBandPairsU returns a pointer to an FNC\_QB\_Pair\_t structure.

FNC\_QB\_Pair\_t is defined in sqltypes\_td.h and has the following members:

Member ...	Specifies ...
QBName	the query band name in UNICODE.
QBValue	the query band value in UNICODE.



FNC\_GetQueryBandPairsU allocates local storage for the return structure. If the returned number of pairs is greater than zero, the UDF, UDM, or external stored procedure must call FNC\_free to free the return structure before exiting.

An external stored procedure that uses CLlv2 to execute SQL must wait for any outstanding CLlv2 requests to complete before calling this function.

## Syntax

```
FNC_QB_Pair_t *
FNC_GetQueryBandPairsU ( void          *QBandBuf,
                        FNC_QBSearch_et SearchType,
                        int             *NumPairs,
                        word            QBCharType );
```

### Syntax Elements

#### ***QBandBuf***

A pointer to a buffer containing the query band, where the query band can be:

- Returned by FNC\_GetQueryBandU
- Retrieved from the DBC.DBQLogTbl.QueryBand column
- Specified by the caller

#### ***SearchType***

Whether FNC\_GetQueryBandPairsU searches for name-value pairs in the transaction, session, and/or profile query band.

FNC\_QBSearch\_et is defined in sqltypes\_td.h and includes the following values:

- QB\_FIRST specifies to return the first unique name-value pair for each name found in the transaction, session, and profile query bands. If the transaction, session, and profile query bands contain the same name, FNC\_GetQueryBandPairsU returns the first name-value pair found in the query band in the following order:
  - Transaction query band
  - Session query band
  - Profile query band
- QB\_TXN specifies to return name-value pairs in the transaction query band.
- QB\_SESSION specifies to return name-value pairs in the session query band.
- QB\_PROFILE specifies to return name-value pairs in the profile query band.

#### ***NumPairs***

The number of name-value pairs that FNC\_GetQueryBandPairsU returns.

**QBCharType**

The character set of the query band string in *QBandBuf*.

Supported values:

- 1 (LATIN)
- 2 (UNICODE)

**Example Using FNC\_GetQueryBandPairsU****SQL Definition**

```
REPLACE FUNCTION GetQBPairsExample()
  RETURNS TABLE
  (QBName      VARCHAR(129) CHARACTER SET UNICODE,
   QBValue     VARCHAR(257) CHARACTER SET UNICODE)
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!GetQBPairsExample!GetQBPairsExample.c';
```

**C Function Definition**

```
#define SQL_TEXT Unicode_Text

#include<string.h>
#include<sqltypes_td.h>

typedef unsigned short int word;
#define EVLLATIN1 1
#define EVLUNICODE 2

extern void
FNC_GetQueryBandU(void      *QBandBuf,
                  int       BufSize,
                  int       *QBandLen);

extern FNC_QB_Pair_t *
FNC_GetQueryBandPairsU (
    void      *QBandBuf,
    FNC_QBSearch_et QB_SearchType,
    int       *numPairs,
    word      QBCharType);
```

```

#define EndofDataState      "02000"
#define InvalidParamState  "22023"
#define InsuffMemoryState  "54001"

typedef struct {
    int cnt;
    int NumPairs;
    FNC_QB_Pair_t *pairPtr;
} CTX;

/* copy source to destination */
void tdwstrcpy(VARCHAR_UNICODE      *dest,
               const VARCHAR_UNICODE *src)
{
    while(*src != (VARCHAR_UNICODE)0)
        *dest++ = *src++;
    *dest = (VARCHAR_UNICODE)0;
}

/*****/

void GetQBPairsExample(
    VARCHAR_UNICODE *QBName,
    VARCHAR_UNICODE *QBValue,
    int *qbname_i,
    int *qbvalue_i,
    char sqlstate[6],
    SQL_TEXT fncname[129],
    SQL_TEXT sfncname[129],
    SQL_TEXT error_message[257] )
{
    int   QBandLen;
    int   minQBLen = 5;
    CTX   *ctx;
    char  *QBandBuf;
    FNC_Phase_en Phase;
    int   MaxQBLen = FNC_MAXQUERYBANDSIZE_U;

    /* Depending on the phase decide what to do. */
    switch (FNC_GetPhase(&Phase))
    {
        /*****/

```

```

/* Process the constant expression case. Only one AMP will */
/* participate for this example */
/*****
case TBL_MODE_CONST:
{
    switch(Phase)
    {
    case TBL_PRE_INIT:
        switch (FNC_TblFirstParticipant() )
        {
        case 1: /* participant */
            return;
        case 0: /* not participant */
            FNC_TblOptOut();
            return;
        default:
            return;
        }
        break;

    case TBL_INIT:
        /* allocate context */
        ctx = FNC_TblAllocCtx(sizeof(CTX));
        ctx->cnt = 0;

        /* allocate buffer for the queryband */
        QBandBuf = FNC_malloc(MaxQBLen);
        if (QBandBuf == NULL)
        {
            strcpy(sqlstate, InsuffMemoryState);
            return;
        }

        /* Get the queryband */
        FNC_GetQueryBandU(QBandBuf, MaxQBLen, &QBandLen);

        ctx->NumPairs = 0;
        if (QBandLen >= minQBLen)
        {
            /* Get the pairs */
            ctx->pairPtr = FNC_GetQueryBandPairsU(QBandBuf, 0,
                &ctx->NumPairs, EVLUNICODE);
        }
        FNC_free(QBandBuf);

```

```

        break;

    case TBL_BUILD:
        /* Read and build result rows */

        ctx = FNC_TblGetCtx();

        if (ctx->cnt < ctx->NumPairs)
        {
            tdwstrcpy(QBName, ctx->pairPtr[ctx->cnt].QBName);
            tdwstrcpy(QBValue, ctx->pairPtr[ctx-
>cnt].QBValue);
            ctx->cnt++;
        }
        else
        {
            /* Have an end of file here. Let the system know. */
            strcpy(sqlstate, EndofDataState);
        }

        break;

    case TBL_END:
    case TBL_ABORT:

        /* Everyone is done. */
        ctx = FNC_TblGetCtx();
        FNC_free(ctx->pairPtr);
        break;

    } /* end switch phase */

} /* end case TBL_MODE_CONST */
break;
/*****
/* Process the varying expression */
*****/
case TBL_MODE_VARY:
{
    switch(Phase)
    {
    case TBL_PRE_INIT:
        strcpy(sqlstate, InvalidParamState);

```

```

        break;
    case TBL_INIT:
    case TBL_BUILD:
    case TBL_FINI:
    case TBL_ABORT:
    case TBL_END:
        break;
    } /* end switch phase */

} /* case TBL_MODE_VARY */
break;
}

return ;
}

```

## FNC\_GetQueryBandValue

Search the transaction, session, and/or profile name-value pairs in the query band string pointed to by the *QBandBuf* input argument and retrieve the value for a specified name. The query band value returned is in the character set of the function (LATIN or UNICODE).

## Syntax

```

void
FNC_GetQueryBandValue ( void          *QBandBuf,
                        FNC_QBSearch_et SearchType
                        void          *QBName,
                        void          *QBValue )

```

## Syntax Elements

### *QBandBuf*

A pointer to a buffer containing the query band, where the query band can be:

- Returned by FNC\_GetQueryBand
- Retrieved from the DBC.DBQLogTbl.QueryBand column
- Specified by the caller

### *SearchType*

Whether FNC\_GetQueryBandValue (or FNC\_GetQueryBandValueU) searches the transaction, session, and/or profile name-value pairs in the query band string for the name specified by *QBName*.

FNC\_QBSearch\_et is defined in sqltypes\_td.h and includes the following values:

- QB\_FIRST specifies to return the value of the first name-value pair, where the name is specified by the *QBName* input argument. If the query band string contains name-value pairs for the transaction, session, and profile, FNC\_GetQueryBandValue (or FNC\_GetQueryBandValueU) searches the name-value pairs in the following order:
  - Transaction query band
  - Session query band
  - Profile query band
- QB\_TXN specifies to search the transaction name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.
- QB\_SESSION specifies to search the session name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.
- QB\_PROFILE specifies to search the profile name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.

### ***QBName***

The name in the name-value pair to return the value for.

### ***QBValue***

A pointer to a buffer in which FNC\_GetQueryBandValue (or FNC\_GetQueryBandValueU) returns the value corresponding to the name specified by *QBName*.

The buffer must be large enough to return the value plus a null terminator.

## **Usage Notes**

The character set of a UDF is determined by the character set of the user that creates the function. Therefore, if the user creating a UDF has a character set of LATIN, the UDF has a character set of LATIN. If the user creating a UDF has a character set of UNICODE, the UDF has a character set of UNICODE. FNC\_GetQueryBandValue assumes the input character parameters are in the function character set and returns the output parameters in the same character set.

## **System Routines that Use FNC\_GetQueryBandValue**

Teradata provides a UDF called GetQueryBandValue and an external stored procedure called GetQueryBandValueSP that use FNC\_GetQueryBandValue to search the transaction, session, and/or profile name-value pairs in the query band and retrieve the value for a specified name. GetQueryBandValue and GetQueryBandValueSP are created in the SYSLIB database when you execute the DIPDEM script using the DIP utility.

For more information on DIP, see *Teradata Vantage™ - Database Utilities*, B035-1102. For more information on GetQueryBandValue and GetQueryBandValueSP, see *Teradata Vantage™ - Application Programming Reference*, B035-1090.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_GetQueryBandValue

```
#define SQL_TEXT Latin_Text
#include<sqltypes_td.h>

void getJobTitle( VARCHAR_LATIN *jobTitle, char sqlstate[6])
{
    char *QBBuf;
    int QBLen;

    QBBuf = FNC_malloc(FNC_MAXQUERYBANDSIZE);
    if (QBBuf == NULL)
    {
        strcpy(sqlstate, "U0004");
        strcpy((char *) error_message, "malloc failed");
    }
    else
    {
        FNC_GetQueryBand(QBBuf, FNC_MAXQUERYBANDSIZE, &QBLen);
        if (QBLen > 0)
        {
            FNC_GetQueryBandValue(QBBuf, QB_FIRST, "JOBTITLE", jobTitle);
        }
        strcpy(sqlstate, "00000");
        FNC_free(QBBuf);
    }
    return;
}
```



## FNC\_GetQueryBandValueU

Search the transaction, session, and/or profile name-value pairs in the query band string pointed to by the *QBandBuf* input argument and retrieve the value for a specified name. The query band value is returned in the UNICODE character set.

An external stored procedure that uses CLlV2 to execute SQL must wait for any outstanding CLlV2 requests to complete before calling this function.

### Syntax

```
void
FNC_GetQueryBandValueU ( void          *QBandBuf,
                        FNC_QBSearch_et SearchType,
                        void          *QBName,
                        void          *QBValue,
                        word          QBCharType,
                        word          NameCharType);
```

### Syntax Elements

#### *QBandBuf*

A pointer to a buffer containing the query band, where the query band can be:

- Returned by FNC\_GetQueryBand
- Retrieved from the DBC.DBQLogTbl.QueryBand column
- Specified by the caller

#### *SearchType*

Whether FNC\_GetQueryBandValue (or FNC\_GetQueryBandValueU) searches the transaction, session, and/or profile name-value pairs in the query band string for the name specified by *QBName*.

FNC\_QBSearch\_et is defined in sqltypes\_td.h and includes the following values:

- QB\_FIRST specifies to return the value of the first name-value pair, where the name is specified by the *QBName* input argument. If the query band string contains name-value pairs for the transaction, session, and profile, FNC\_GetQueryBandValue (or FNC\_GetQueryBandValueU) searches the name-value pairs in the following order:
  - Transaction query band
  - Session query band
  - Profile query band
- QB\_TXN specifies to search the transaction name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.

- `QB_SESSION` specifies to search the session name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.
- `QB_PROFILE` specifies to search the profile name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.

***QBName***

The name in the name-value pair to return the value for.

***QBValue***

A pointer to a buffer in which `FNC_GetQueryBandValue` (or `FNC_GetQueryBandValueU`) returns the value corresponding to the name specified by *QBName*.

The buffer must be large enough to return the value plus a null terminator.

***QBCharType***

The character set of the query band string in *QBandBuf*.

Supported values:

- 1 (LATIN)
- 2 (UNICODE)

***NameCharType***

The character set of the query band name in *QBName*.

Supported values:

- 1 (LATIN)
- 2 (UNICODE)

## Example Using `FNC_GetQueryBandValueU`

This example shows a UDF that accepts a LATIN or UNICODE input name parameter.

**SQL Definition**

```
REPLACE FUNCTION GetQBValueExample(
    SearchType SMALLINT,
    QBName      TD_ANYTYPE)
RETURNS VARCHAR(257) CHARACTER SET UNICODE
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
EXTERNAL NAME 'CS!GetQBValExample!GetQBValExample.c';
```

**C Function Definition**

```

#define SQL_TEXT Latin_Text

#include<string.h>
#include<sqltypes_td.h>

typedef unsigned short int word;
#define EVLLATIN1 1
#define EVLUNICODE 2

extern void
FNC_GetQueryBandU(void      *QBAndBuf,
                  int        BufSize,
                  int        *QBAndLen);

extern void
FNC_GetQueryBandValueU (
    void      *QBAndBuf,
    FNC_QBSearch_et QB_SearchType,
    void      *QBName,
    void      *QBValue,
    word      QBCharType,
    word      NameCharType);

#define OKState          "00000"
#define InvalidParamState "22023"
#define InsuffMemoryState "54001"

/******/

void GetQBValueExample(
    void      *QBName,
    VARCHAR_UNICODE *QBValue,
    char sqlstate[6])
{
    int QBAndLen;
    int minQBLen = 5;
    VARCHAR_UNICODE *QBAndBuf;
    word NameCharType;
    int numAnytypeParams;
    anytype_param_info_t inputInfo[1];

```

```

int MaxQBLen;

MaxQBLen = FNC_MAXQUERYBANDSIZE_U;

/* QBName is TD_ANYTYPE */
FNC_GetAnyTypeInfo(sizeof(anytype_param_info_t),
    &numAnytypeParams, inputInfo);
if ((numAnytypeParams >= 0) &&
    (inputInfo[0].paramIndex == 1) &&
    ((inputInfo[0].datatype == VARCHAR_DT) ||
    (inputInfo[0].datatype == CHAR_DT))
    )
{
    if (inputInfo[0].charset == LATIN_CT)
        NameCharType = EVLLATIN1;
    else
        NameCharType = EVLUNICODE;
}
else
{
    strcpy(sqlstate, InvalidParamState);
    return;
}

/* allocate buffer for the queryband */
QBandBuf = FNC_malloc(MaxQBLen);
if (QBandBuf == NULL)
{
    strcpy(sqlstate, InsuffMemoryState);
    return;
}

/* Get the queryband */
FNC_GetQueryBandU(QBandBuf, MaxQBLen, &QBandLen);
if (QBandLen >= minQBLen)
{
    /* Get the value */
    FNC_GetQueryBandValueU(QBandBuf, 0, QBName, QBValue,
        EVLUNICODE, NameCharType);
}
else
{
    /* Return an empty string for the value */
    memcpy(QBValue, "\0\0", 2);
}

```

```

    }

    FNC_free(QBandBuf);
    strcpy(sqlstate, OKState);

    return ;
}

```

## FNC\_GetStructuredAttribute

Returns the value of an attribute of a structured type that is defined to be an input parameter to a UDF, UDM, or external stored procedure.

The attribute must be one of the following types:

- Non-LOB predefined type, such as INTEGER or VARCHAR
- Distinct UDT
- Structured UDT
- JSON

To get the value of a LOB attribute of a structured type, use the C library routine [FNC\\_GetStructuredInputLobAttributeByNdx](#).

---

### Note:

For best performance, use [FNC\\_SetStructuredAttributeByNdx](#) instead of this routine. This routine is supported for ease of use.

---

## Syntax

```

void
FNC_GetStructuredAttribute ( UDT_HANDLE  udtHandle,
                           char          *attributePath,
                           void          *returnValue,
                           int           bufSize,
                           int           *nullIndicator,
                           int           *length )

```

### Syntax Elements

#### *udtHandle*

the handle to a structured UDT that is defined to be an input parameter to a UDF, UDM, or external stored procedure.

***attributePath***

the dot delimited full path to the attribute.

For example, consider a structured UDT called "PersonUDT" that has an attribute called "address" that is an AddressUDT type, which in turn has an attribute called "zipcode". To get the zipcode value, the full path is "address.zipcode".

***returnValue***

a pointer to a buffer that FNC\_GetStructuredAttribute uses to return the value of the attribute.

***bufSize***

the size in bytes of the *returnValue* buffer.

***nullIndicator***

whether the attribute is null.

If the value of *nullIndicator* is...

- -1, then the attribute is null.

A value of -1 can also indicate that the full path to the specified attribute includes a preceding null attribute.

- 0, then the attribute is not null.

***length***

the size in bytes of the value that FNC\_GetStructuredAttribute returns in *returnValue*.

For character data types, the length includes the size of any null termination characters.

## Usage Notes

To get a string representation of a JSON attribute, do the following:

1. Call FNC\_GetStructuredAttributeInfo\_EON.
2. If the *lob\_length* returned from the previous function call is 0, you can use FNC\_GetStructuredAttribute to get a string representation of a JSON attribute.
3. If the *lob\_length* is not 0, you must use FNC\_GetStructuredInputLobAttribute instead of FNC\_GetStructuredAttribute.

Before calling FNC\_GetStructuredAttribute, you must allocate the buffer pointed to by *returnValue* using the C data type that maps to the underlying type of the attribute. For example, if the underlying type of the attribute is an SQL INTEGER data type, declare the buffer like this:

```
INTEGER value;
```

If the attribute is a distinct type, allocate a buffer using the C data type that the distinct type represents. For example, if the attribute is a distinct type that represents an SQL SMALLINT data type, declare the buffer like this:

```
SMALLINT value;
```

If the underlying type of the attribute is a character string, FNC\_GetStructuredAttribute returns a null-terminated character string. The buffer you define must be large enough to accommodate null termination.

If the attribute is a structured type, FNC\_GetStructuredAttribute returns a UDT handle. To get descriptive information on the attributes of the structured type, pass the UDT handle to FNC\_GetStructuredAttributeInfo.

For information on the C data types that you can use, see [C Data Types](#).

To guarantee that the value you pass in for the *bufSize* argument matches the length of the data type, use the following macros defined in the `sqltypes_td.h` header file:

Macro	Description
SIZEOF_CHARACTER_LATIN_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_KANJISJIS_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_KANJI1_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_UNICODE_WITH_NULL( <i>len</i> )	Returns the length in bytes of a CHARACTER data type of <i>len</i> characters, including null termination characters. For example, the following returns a length of 8 ( $3 * 2 + 2 = 8$ ):  SIZEOF_CHARACTER_UNICODE_WITH_NULL(3)
SIZEOF_VARCHAR_LATIN_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_KANJISJIS_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_KANJI1_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_UNICODE_WITH_NULL( <i>len</i> )	Returns the length in bytes of a VARCHAR data type of <i>len</i> characters, including null termination characters. For example, the following returns a length of 8 ( $3 * 2 + 2 = 8$ ):  SIZEOF_VARCHAR_UNICODE_WITH_NULL(3)
SIZEOF_BYTE( <i>len</i> ) SIZEOF_VARBYTE( <i>len</i> )	Returns the length in bytes of the specified BYTE or VARBYTE data type, where <i>len</i> specifies the number of values.
SIZEOF_GRAPHIC( <i>len</i> ) SIZEOF_VARGRAPHIC( <i>len</i> )	Returns the length in bytes of the specified CHARACTER(n) CHARACTER SET GRAPHIC or VARCHAR(n) CHARACTER SET GRAPHIC data type, where <i>len</i> specifies the number of values.
SIZEOF_BYTEINT SIZEOF_SMALLINT SIZEOF_INTEGER SIZEOF_BIGINT SIZEOF_REAL SIZEOF_DOUBLE_PRECISION	Returns the length in bytes of the specified numeric data type.  For NUMBER, the length returned is $4 + 2 + 17 = 23$ bytes since Vantage allocates max length (17 bytes) for the mantissa.

Macro	Description
SIZEOF_FLOAT SIZEOF_DECIMAL1 SIZEOF_DECIMAL2 SIZEOF_DECIMAL4 SIZEOF_DECIMAL8 SIZEOF_DECIMAL16 SIZEOF_NUMERIC1 SIZEOF_NUMERIC2 SIZEOF_NUMERIC4 SIZEOF_NUMERIC8 SIZEOF_NUMERIC16 SIZEOF_NUMBER	
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.
SIZEOF_INTERVAL_YEAR SIZEOF_IntrvlYtoM SIZEOF_INTERVAL_MONTH SIZEOF_INTERVAL_DAY SIZEOF_IntrvlDtoH SIZEOF_IntrvlDtoM SIZEOF_IntrvlDtoS SIZEOF_HOUR SIZEOF_IntrvlHtoM SIZEOF_IntrvlHtoS SIZEOF_MINUTE SIZEOF_IntrvlMtoS SIZEOF_IntrvlSec	Returns the length in bytes of the specified Interval type.
SIZEOF_UDT_HANDLE	Returns the length in bytes of a handle to a UDT.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_GetStructuredAttribute

```
void getX( UDT_HANDLE *pointUdt,
          INTEGER      *result,
          char         sqlstate[6])
```



```

{
    INTEGER x;
    int nullIndicator;
    int length;

    /* Get the x attribute of pointUdt. */
    FNC_GetStructuredAttribute(*pointUdt, "x", &x, sizeof_INTEGER,
                              &nullIndicator, &length);

    if (nullIndicator == -1) {
        /* do null handling here */
        ...
        return;
    }

    ...
}

```

## FNC\_GetStructuredAttributeByNdx

Returns the value of a non-LOB attribute of a structured UDT.

The attribute must be one of the following types:

- Non-LOB predefined type, such as INTEGER or VARCHAR
- Distinct UDT
- Structured UDT

To get the value of a LOB attribute of a structured type, use one of the following:

- [FNC\\_GetStructuredInputLobAttributeByNdx](#)
- [FNC\\_GetStructuredResultLobAttributeByNdx](#)

## Syntax

```

void
FNC_GetStructuredAttributeByNdx ( UDT_HANDLE  udtHandle,
                                int           attributeIndex,
                                void          *returnValue,
                                int           bufSize,
                                int           *nullIndicator,
                                int           *length )

```

## Syntax Elements

### *udtHandle*

The handle to a structured UDT:

### *attributeIndex*

The index of an attribute at the top-most level of the UDT.

The range of values is from 0 to  $i-1$ , where 0 is the index of the first attribute in the UDT and  $i$  is the number of non-nested attributes in the UDT.

For example, consider a structured UDT called PointUDT that has two attributes: the first attribute is called x and the second attribute is called y. To get the x value, use an index of 0. Similarly, to get the y value, use an index of 1.

For details on how to get the value of a nested attribute, see [Getting the Value of a Nested Attribute](#).

### *returnValue*

A pointer to a buffer that FNC\_GetStructuredAttributeByNdx uses to return the value of the attribute.

### *bufSize*

the size in bytes of the *returnValue* buffer.

### *nullIndicator*

Whether the attribute is null.

If the value of *nullIndicator* is...

- -1, then the attribute is null.  
A value of -1 can also indicate that the *udtHandle* argument is null.
- 0, then the attribute is not null.

### *length*

The size in bytes of the value that FNC\_GetStructuredAttributeByNdx returns in *returnValue*.

For character data types, the length includes the size of any null termination characters.

## Usage Notes

Before calling FNC\_GetStructuredAttributeByNdx, you must allocate the buffer pointed to by *returnValue* using the C data type that maps to the underlying type of the attribute. For example, if the underlying type of the attribute is an SQL INTEGER data type, declare the buffer like this:

```
INTEGER value;
```

If the attribute is a distinct type, allocate a buffer using the C data type that the distinct type represents. For example, if the attribute is a distinct type that represents an SQL SMALLINT data type, declare the buffer like this:

```
SMALLINT value;
```

If the underlying type of the attribute is a character string, `FNC_GetStructuredAttributeByNdx` returns a null-terminated character string. The buffer you define must be large enough to accommodate null termination.

If the attribute is a structured type, `FNC_GetStructuredAttributeByNdx` returns a UDT handle. To get descriptive information on the attributes of the structured type, pass the UDT handle to `FNC_GetStructuredAttributeInfo`.

For information on the C data types that you can use, see [C Data Types](#).

To guarantee that the value you pass in for the `bufSize` argument matches the length of the data type, use the following macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_CHARACTER_LATIN_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_KANJISJIS_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_KANJI1_WITH_NULL( <i>len</i> ) SIZEOF_CHARACTER_UNICODE_WITH_NULL( <i>len</i> )	Returns the length in bytes of a CHARACTER data type of <i>len</i> characters, including null termination characters. For example, the following returns a length of 8 ( $3 * 2 + 2 = 8$ ): SIZEOF_CHARACTER_UNICODE_WITH_NULL(3)
SIZEOF_VARCHAR_LATIN_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_KANJISJIS_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_KANJI1_WITH_NULL( <i>len</i> ) SIZEOF_VARCHAR_UNICODE_WITH_NULL( <i>len</i> )	Returns the length in bytes of a VARCHAR data type of <i>len</i> characters, including null termination characters. For example, the following returns a length of 8 ( $3 * 2 + 2 = 8$ ): SIZEOF_VARCHAR_UNICODE_WITH_NULL(3)
SIZEOF_BYTE( <i>len</i> ) SIZEOF_VARBYTE( <i>len</i> )	Returns the length in bytes of the specified BYTE or VARBYTE data type, where <i>len</i> specifies the number of values.
SIZEOF_GRAPHIC( <i>len</i> ) SIZEOF_VARGRAPHIC( <i>len</i> )	Returns the length in bytes of the specified CHARACTER(n) CHARACTER SET GRAPHIC or VARCHAR(n) CHARACTER SET GRAPHIC data type, where <i>len</i> specifies the number of values.
SIZEOF_BYTEINT SIZEOF_SMALLINT	Returns the length in bytes of the specified numeric data type.

Macro	Description
SIZEOF_INTEGER SIZEOF_BIGINT SIZEOF_REAL SIZEOF_DOUBLE_PRECISION SIZEOF_FLOAT SIZEOF_DECIMAL1 SIZEOF_DECIMAL2 SIZEOF_DECIMAL4 SIZEOF_DECIMAL8 SIZEOF_DECIMAL16 SIZEOF_NUMERIC1 SIZEOF_NUMERIC2 SIZEOF_NUMERIC4 SIZEOF_NUMERIC8 SIZEOF_NUMERIC16 SIZEOF_NUMBER	For NUMBER, the length returned is $4 + 2 + 17 = 23$ bytes since Vantage allocates max length (17 bytes) for the mantissa.
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.
SIZEOF_INTERVAL_YEAR SIZEOF_IntrvlytoM SIZEOF_INTERVAL_MONTH SIZEOF_INTERVAL_DAY SIZEOF_IntrvldtoH SIZEOF_IntrvldtoM SIZEOF_IntrvldtoS SIZEOF_HOUR SIZEOF_IntrvlHtoM SIZEOF_IntrvlHtoS SIZEOF_MINUTE SIZEOF_IntrvlMtoS SIZEOF_IntrvlSec	Returns the length in bytes of the specified Interval type.
SIZEOF_UDT_HANDLE	Returns the length in bytes of a handle to a UDT.

## Getting the Value of a Nested Attribute

To get the value of a nested attribute in a structured UDT, follow these steps:

1. Call `FNC_GetStructuredAttributeByNdx` to obtain the UDT handle of the next structured UDT attribute in the hierarchy.

Repeat this step, passing in the newly obtained UDT handle as the *udtHandle* argument,  $n-2$  times, where  $n$  is the nesting level of the target attribute.

2. Call `FNC_GetStructuredAttributeByNdx` again, passing in the UDT handle of the structured UDT attribute that contains the target attribute.

## Restrictions

An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

## Example Using `FNC_GetStructuredAttributeByNdx`

```
void getY( UDT_HANDLE *pointUdt,
          INTEGER    *result,
          char        sqlstate[6])
{
    INTEGER y;
    int nullIndicator;
    int length;

    /* Get the y attribute (second attribute) of pointUdt. */
    FNC_GetStructuredAttributeByNdx(*pointUdt, 1, &y, sizeof_INTEGER,
    &nullIndicator, &length);
    if (nullIndicator == -1) {
        /* do null handling here */
        ...
        return;
    }

    ...
}
```

## `FNC_GetStructuredAttributeCount`

Returns the total number of attributes for a structured type that is defined to be an input parameter to an external routine.

## Syntax

```
void
FNC_GetStructuredAttributeCount ( UDT_HANDLE  udtHandle,
                                int          *attributeCount )
```

## Syntax Elements

### *udtHandle*

the handle to a structured UDT that is defined to be an input parameter to an external routine.

### *attributeCount*

the number of attributes for the structured UDT specified by *udtHandle*.

An attribute that is a structured UDT only counts as one attribute.

## Usage Notes

You can use `FNC_GetStructuredAttributeCount` to get the number of attributes of any structured UDT, including dynamic UDTs that are defined as input parameters to UDFs.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using `FNC_GetStructuredAttributeCount`

```
void getMap( UDT_HANDLE *geometriesUdt,
            INTEGER    *result,
            char        sqlstate[6])
{
    int geometriesCount;

    /* Get the number of attributes of geometriesUdt. */
    FNC_GetStructuredAttributeCount((*geometriesUdt), &geometriesCount);

    ...
}
```

## `FNC_GetStructuredAttributeInfo` [Deprecated]

This function is deprecated because it truncates object names to 30 characters. However, it remains available to support legacy applications. For current and future development, use the corresponding function that includes "EON" in the function name. For example, use `FNC_DbsInfo_EON` instead of `FNC_DbsInfo`.

Returns information, such as data type, about the attributes of a structured type that is defined to be an input parameter to an external routine.

## Syntax

```
void
FNC_GetStructuredAttributeInfo ( UDT_HANDLE      udtHandle,
                                int              attributePosition,
                                int              bufSize,
                                attribute_info_t *attributeArray )
```

### ***attributeArray***

Defined in `sqltypes_td.h` as:

```
typedef struct attribute_info_t
{
    INTEGER      attrIndex;
    dtype_et     data_type;
    CHARACTER    attribute_name[256];
    SMALLINT     udt_indicator;
    CHARACTER    udt_type_name[256];
    INTEGER      max_length;
    FNC_LobLength_t lob_length;
    SMALLINT     total_interval_digits;
    SMALLINT     num_fractional_digits;
    charset_et   charset_code;
} attribute_info_t;
```

## Syntax Elements

### ***udtHandle***

Handle to a structured UDT that is defined to be an input parameter to an external routine.

### ***attributePosition***

Whether this call returns information about one or all of the attributes:

<b><i>attributePosition</i></b>	<b>Description</b>
0 or greater	Value identifies position of attribute in structured UDT for which this call returns information. Position of first attribute is 0.
-1	Call returns information about all attributes.

**bufSize**

Size in bytes that was allocated to the *attributeArray* argument.

**attributeArray**

Array of one or more `attribute_info_t` structures that describe the requested attribute or attributes in the structured UDT, as specified by *attributePosition*.

**attrIndex**

Specifies the position of the attribute in the structured type, where the position of the first attribute is 0.

**data\_type**

Specifies the data type of the attribute. The `sqltypes_td.h` header file defines `dtype_et` as:

```
typedef int dtype_et;
```

Valid values are defined by the `dtype_en` enumeration in `sqltypes_td.h`. For a list of the valid values, see the *dtype* argument in [FNC\\_CallSP](#).

**attribute\_name**

Specifies the attribute name.

For dynamic UDTs with attributes that are defined by column names and no , the attribute name is the column name.

**udt\_indicator**

Specifies whether the attribute is a UDT and if so, which kind:

<i>udt_indicator</i>	UDT Kind
0	Not a UDT
1	Structured UDT
2	Distinct UDT
3	Teradata proprietary internal UDT
5	JSON attribute

**udt\_type\_name**

Specifies the UDT type name associated with the attribute. Meaningful only if the attribute is a UDT.



***max\_length***

Specifies the maximum length in bytes of the value of the attribute:

Attribute Type	<i>max_length</i>
Non-LOB	Size in bytes of buffer you must allocate before calling FNC_GetStructuredAttribute to get attribute value.
LOB	LOB_REF length <i>lob_length</i> provides length of LOB data itself.
JSON	Maximum length of JSON attribute.

***lob\_length***

Specifies the length of a LOB attribute. Meaningful only if the attribute is a LOB.

For a JSON attribute, *lob\_length* specifies the maximum possible length of the JSON attribute in bytes if the data is stored as a LOB; otherwise, this value is 0.

You can use *lob\_length* as the length of the LOB object that you pass to FNC\_LobOpen when you establish a read context for the LOB attribute.

***total\_interval\_digits***

Specifies the precision of certain attributes. The value of *total\_interval\_digits* corresponds to the *n* value of the following types:

- DECIMAL(*n*, *m*)
- INTERVAL YEAR(*n*)
- INTERVAL YEAR(*n*) TO MONTH
- INTERVAL MONTH(*n*)
- INTERVAL DAY(*n*)
- INTERVAL DAY(*n*) TO HOUR
- INTERVAL DAY(*n*) TO MINUTE
- INTERVAL DAY(*n*) TO SECOND(*m*)
- INTERVAL HOUR(*n*)
- INTERVAL HOUR(*n*) TO MINUTE
- INTERVAL HOUR(*n*) TO SECOND(*m*)
- INTERVAL MINUTE(*n*)
- INTERVAL MINUTE(*n*) TO SECOND(*m*)
- INTERVAL SECOND(*n*, *m*)

If the data type of the attribute does not appear in the preceding list, *total\_interval\_digits* is not meaningful for the attribute.

***num\_fractional\_digits***

Specifies the precision or scale of certain attributes. The value of *num\_fractional\_digits* corresponds to the *m* value of the following types:

- DECIMAL(*n*, *m*)
- TIME(*m*)
- TIME(*m*) WITH TIME ZONE
- TIMESTAMP(*m*)
- TIMESTAMP(*m*) WITH TIME ZONE
- INTERVAL DAY(*n*) TO SECOND(*m*)
- INTERVAL HOUR(*n*) TO SECOND(*m*)
- INTERVAL MINUTE(*n*) TO SECOND(*m*)
- INTERVAL SECOND(*n*, *m*)

If the data type of the attribute does not appear in the preceding list, *num\_fractional\_digits* is not meaningful for the attribute.

***charset\_code***

Specifies the server character set associated with the attribute. Meaningful only if the attribute is a character type.

The `sqltypes_td.h` header file defines `charset_et` as:

```
typedef int charset_et;
```

Valid values are defined by the `charset_en` enumeration in `sqltypes_td.h`:

```
typedef enum charset_en
{
    UNDEF_CT=0,
    LATIN_CT=1,
    UNICODE_CT=2,
    KANJISJIS_CT=3,
    KANJI1_CT=4
} charset_en;
```

For a JSON attribute, *charset\_code* is `LATIN_CT` or `UNICODE_CT` depending on how the JSON attribute was defined.

## Usage Notes

You can use `FNC_GetStructuredAttributeInfo` but it truncates a retrieved object name after 30 characters. Use `FNC_GetStructuredAttributeInfo_EON` instead of `FNC_GetStructuredAttributeInfo` for longer object names.

You can use `FNC_GetStructuredAttributeInfo` to get information about the attributes of any structured UDT, including dynamic UDTs that are defined as input parameters to UDFs.

## Restrictions

An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

## Example Using `FNC_GetStructuredAttributeInfo`

```
void getMap( UDT_HANDLE *geometriesUdt,
            INTEGER      *result,
            char          sqlstate[6])
{
    attribute_info_t attributeInfo;
    int nullIndicator;
    int length;
    int bufSize = 0;
    void * tmpBuf = 0;

    /* Get the attribute information for the first attribute. */
    FNC_GetStructuredAttributeInfo(*geometriesUdt, 0,
        sizeof(attribute_info_t), &attributeInfo);
    /* Get the value of the first attribute. */
    bufSize = attributeInfo.max_length;
    tmpBuf = FNC_malloc(bufSize);
    FNC_GetStructuredAttributeByNdx(*geometriesUdt, 0, tmpBuf, bufSize,
        &nullIndicator, &length);

    ...
}
```

## `FNC_GetStructuredAttributeInfo_EON`

Returns information, such as data type, about the attributes of a structured type that is defined to be an input parameter to an external routine.

## Syntax

```
void
FNC_GetStructuredAttributeInfo_EON ( UDT_HANDLE          AnyStructuredUdt,
                                     int                  attribute_position,
```

```

int          bufSize,
attribute_info_eon_t *attributeArray )

```

**attributeArray**

Defined in sqltypes\_td.h as:

```

typedef struct attribute_info_eon_t
{
    INTEGER          attrIndex;
    dtype_et         data_type;
    CHARACTER        attribute_name[FNC_MAXNAMELEN_EON];
    SMALLINT         udt_indicator;
    CHARACTER        udt_type_name[FNC_MAXNAMELEN_EON];
    INTEGER          max_length;
    FNC_LobLength_t  lob_length;
    SMALLINT         total_interval_digits;
    SMALLINT         num_fractional_digits;
    charset_et       charset_code;
} attribute_info_eon_t;

```

**Syntax Elements****AnyStructuredUdt**

Handle to a structured UDT that is defined to be an input parameter to an external routine.

**attributePosition**

Whether this call returns information about one or all of the attributes:

<i>attributePosition</i>	Description
0 or greater	Value identifies position of attribute in structured UDT for which this call returns information. Position of first attribute is 0.
-1	Call returns information about all attributes.

**bufSize**

Size in bytes that was allocated to the *attributeArray* argument.

**attributeArray**

Array of one or more *attribute\_info\_eon\_t* structures that describe the requested attribute or attributes in the structured UDT, as specified by *attributePosition*.

***attrIndex***

Specifies the position of the attribute in the structured type, where the position of the first attribute is 0.

***data\_type***

Specifies the data type of the attribute. The `sqltypes_td.h` header file defines `dtype_et` as:

```
typedef int dtype_et;
```

Valid values are defined by the `dtype_en` enumeration in `sqltypes_td.h`. For a list of the valid values, see the *dtype* argument in [FNC\\_CallSP](#).

***attribute\_name***

Specifies the attribute name.

For dynamic UDTs with attributes that are defined by column names and no , the attribute name is the column name.

***udt\_indicator***

Specifies whether the attribute is a UDT and if so, which kind:

<i>udt_indicator</i>	UDT Kind
0	Not a UDT
1	Structured UDT
2	Distinct UDT
3	Teradata proprietary internal UDT
5	JSON attribute

***udt\_type\_name***

Specifies the UDT type name associated with the attribute. Meaningful only if the attribute is a UDT.

***max\_length***

Specifies the maximum length in bytes of the value of the attribute:

Attribute Type	<i>max_length</i>
Non-LOB	Size in bytes of buffer you must allocate before calling <code>FNC_GetStructuredAttribute</code> to get attribute value.
LOB	LOB_REF length

Attribute Type	<i>max_length</i>
	<i>lob_length</i> provides length of LOB data itself.
JSON	Maximum length of JSON attribute.

***lob\_length***

Specifies the length of a LOB attribute. Meaningful only if the attribute is a LOB.

For a JSON attribute, *lob\_length* specifies the maximum possible length of the JSON attribute in bytes if the data is stored as a LOB; otherwise, this value is 0.

You can use *lob\_length* as the length of the LOB object that you pass to FNC\_LobOpen when you establish a read context for the LOB attribute.

***total\_interval\_digits***

Specifies the precision of certain attributes. The value of *total\_interval\_digits* corresponds to the *n* value of the following types:

- DECIMAL(*n*, *m*)
- INTERVAL YEAR(*n*)
- INTERVAL YEAR(*n*) TO MONTH
- INTERVAL MONTH(*n*)
- INTERVAL DAY(*n*)
- INTERVAL DAY(*n*) TO HOUR
- INTERVAL DAY(*n*) TO MINUTE
- INTERVAL DAY(*n*) TO SECOND(*m*)
- INTERVAL HOUR(*n*)
- INTERVAL HOUR(*n*) TO MINUTE
- INTERVAL HOUR(*n*) TO SECOND(*m*)
- INTERVAL MINUTE(*n*)
- INTERVAL MINUTE(*n*) TO SECOND(*m*)
- INTERVAL SECOND(*n*, *m*)

If the data type of the attribute does not appear in the preceding list, *total\_interval\_digits* is not meaningful for the attribute.

***num\_fractional\_digits***

Specifies the precision or scale of certain attributes. The value of *num\_fractional\_digits* corresponds to the *m* value of the following types:

- DECIMAL(*n*, *m*)
- TIME(*m*)

- TIME(*m*) WITH TIME ZONE
- TIMESTAMP(*m*)
- TIMESTAMP(*m*) WITH TIME ZONE
- INTERVAL DAY(*n*) TO SECOND(*m*)
- INTERVAL HOUR(*n*) TO SECOND(*m*)
- INTERVAL MINUTE(*n*) TO SECOND(*m*)
- INTERVAL SECOND(*n*, *m*)

If the data type of the attribute does not appear in the preceding list, *num\_fractional\_digits* is not meaningful for the attribute.

### ***charset\_code***

Specifies the server character set associated with the attribute. Meaningful only if the attribute is a character type.

The `sqltypes_td.h` header file defines `charset_et` as:

```
typedef int charset_et;
```

Valid values are defined by the `charset_en` enumeration in `sqltypes_td.h`:

```
typedef enum charset_en
{
    UNDEF_CT=0,
    LATIN_CT=1,
    UNICODE_CT=2,
    KANJISJIS_CT=3,
    KANJI1_CT=4
} charset_en;
```

For a JSON attribute, *charset\_code* is `LATIN_CT` or `UNICODE_CT` depending on how the JSON attribute was defined.

## **Usage Notes**

You can use `FNC_GetStructuredAttributeInfo_EON` to get information about the attributes of any structured UDT, including dynamic UDTs that are defined as input parameters to UDFs.

## **Restrictions**

An external stored procedure that uses `CLIV2` to execute SQL must wait for any outstanding `CLIV2` requests to complete before calling this function.

## FNC\_GetStructuredInputLobAttribute

Returns the locator of a LOB attribute of a structured type that is defined to be an input parameter to a UDF, UDM, or external stored procedure.

---

### Note:

For best performance, use [FNC\\_GetStructuredInputLobAttributeByNdx](#) instead of this routine. This routine is supported for its ease of use.

---

### Syntax

```
void
FNC_GetStructuredInputLobAttribute ( UDT_HANDLE    udtHandle,
                                     char          *attributePath,
                                     int           *nullIndicator,
                                     LOB_LOCATOR  *object )
```

### Syntax Elements

#### *udtHandle*

the handle to a structured UDT that is defined to be an input parameter to a UDF, UDM, or external stored procedure.

#### *attributePath*

the dot delimited full path to the LOB attribute.

For example, consider a structured UDT called "PersonUDT" that has an attribute called "passport" that is a PassportUDT type, which in turn has a LOB attribute called "photo". To get the locator of the photo attribute, the full path is "passport.photo".

#### *nullIndicator*

whether the attribute is null.

If the value of *nullIndicator* is...

- -1, then the attribute is null.  
A value of -1 can also indicate that the full path to the specified attribute includes a preceding null attribute.
- 0, then the attribute is not null.

#### *object*

a pointer to the LOB locator for the LOB attribute.



## Usage Notes

- You can use `FNC_GetStructuredInputLobAttribute` to obtain the LOB locator for a distinct type attribute that represents a LOB type.
- After you obtain the LOB locator for the LOB attribute, use the LOB access functions, such as `FNC_LobOpen`, to read the data.
- To read the data of a JSON attribute which has its data stored in a LOB, do the following:
  1. Call `FNC_GetStructuredAttributeInfo_EON`.
  2. If the `lob_length` returned from the previous function call is 0, you must use `FNC_GetStructuredAttribute` instead of `FNC_GetStructuredInputLobAttribute`.
  3. If the `lob_length` is not 0, use `FNC_GetStructuredInputLobAttribute` to get a `LOB_LOCATOR` which represents the LOB where the data of a JSON attribute is stored. You can use this `LOB_LOCATOR` with the LOB FNC routines to read the data from the JSON attribute.
- An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using `FNC_GetStructuredInputLobAttribute`

```
void document_t_lowerCase( UDT_HANDLE *documentUdt,
                          UDT_HANDLE *resultDocumentUdt,
                          char         sqlstate[6])
{
    LOB_LOCATOR inDocLoc;
    int nullIndicator;

    /* Get a LOB_LOCATOR for the input doc attribute */
    FNC_GetStructuredInputLobAttribute(*documentUdt,
    "doc",                               &nullIndicator, &inDocLoc);    if
    (nullIndicator == -1) {
        /* the CLOB attribute is set to null if we don't append to it. */
        return;
    }
    ....
}
```

## `FNC_GetStructuredInputLobAttributeByNdx`

Returns the locator of a LOB attribute of a structured type that is defined to be an input parameter to an external routine.

## Syntax

```
void
FNC_GetStructuredInputLobAttributeByNdx ( UDT_HANDLE    udtHandle,
                                           int           attributeIndex,
                                           int           *nullIndicator,
                                           LOB_LOCATOR  *object )
```

## Syntax Elements

### *udtHandle*

the handle to a structured UDT that is defined to be one of the following:

- An input parameter to an external routine
- An attribute of a structured UDT that is defined as an input parameter to an external routine

### *attributeIndex*

the index of a non-nested LOB attribute within the UDT.

The range of values is from 0 to  $i-1$ , where 0 is the index of the first attribute in the UDT and  $i$  is the number of top-most level attributes in the UDT.

For example, consider a structured UDT called `passportUDT` that has two attributes: the first attribute is an INTEGER attribute called `ID` and the second attribute is a LOB attribute called `photo`. To get the locator of the `photo` attribute, use an index of 1.

### *nullIndicator*

whether the attribute is null.

If the value of *nullIndicator* is...

- -1, then the attribute is null.  
This can also indicate that the *udtHandle* argument is null.
- 0, then the attribute is not null.

### *object*

a pointer to the LOB locator for the LOB attribute.

## Usage Notes

- You can use `FNC_GetStructuredInputLobAttributeByNdx` to obtain the LOB locator for a distinct type attribute that represents a LOB type.

- After you obtain the LOB locator for the LOB attribute, use the LOB access functions, such as `FNC_LobOpen`, to read the data.
- To get the locator of a nested LOB attribute in a structured UDT that is defined to be an input parameter to an external routine, follow these steps:
  1. Call `FNC_GetStructuredAttributeByNdx` to obtain the UDT handle of the next structured UDT attribute in the path to the target attribute.  
Repeat this step, passing in the newly obtained UDT handle as the *udtHandle* argument, *n-2* times, where *n* is the nesting level of the target attribute.
  2. Call `FNC_GetStructuredInputLobAttributeByNdx`, passing in the UDT handle of the structured UDT attribute that contains the target LOB attribute.
- An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

### Example Using `FNC_GetStructuredInputLobAttributeByNdx`

```
void document_t_lowerCase( UDT_HANDLE *documentUdt,
                          UDT_HANDLE *resultDocumentUdt,
                          char         sqlstate[6])
{
    LOB_LOCATOR inDocLoc;
    int nullIndicator;

    /* Get a LOB_LOCATOR for the input doc (first) attribute */
    FNC_GetStructuredInputLobAttributeByNdx(*documentUdt,
0,                                     &nullIndicator, &inDocLoc);    if
(nullIndicator == -1) {
        /* the CLOB attribute is set to null if we don't append to it. */
        return;
    }
    ....
}
```

## FNC\_GetStructuredResultLobAttribute

Returns the locator of a LOB attribute of a structured type that is defined to be the return value of a UDF or UDM, or an INOUT or OUT parameter to an external stored procedure.

### Note:

For best performance, use [FNC\\_GetStructuredResultLobAttributeByNdx](#) instead of this routine. This routine is supported for ease of use.

## Syntax

```
void
FNC_GetStructuredResultLobAttribute ( UDT_HANDLE      udtHandle,
                                     char             *attributePath,
                                     LOB_RESULT_LOCATOR *object )
```

## Syntax Elements

### *udtHandle*

the handle to a structured UDT that is defined to be the return value of a UDF or UDM or an INOUT or OUT parameter to an external stored procedure.

### *attributePath*

the dot delimited full path to the LOB attribute.

For example, consider a structured UDT called PersonUDT that has an attribute called passport that is a PassportUDT type, which in turn has a LOB attribute called photo. To get the locator of the photo attribute, the full path is "passport.photo".

### *object*

a pointer to the LOB locator for the LOB attribute.

## Usage Notes

You can use FNC\_GetStructuredResultLobAttribute to obtain the LOB locator for a distinct type attribute that represents a LOB type.

After you obtain the LOB locator for the attribute of the structured type, use the LOB access functions, such as FNC\_LobOpen, to append data.

You can use FNC\_GetStructuredResultLobAttribute to write data to a JSON attribute. This function returns a LOB\_RESULT\_LOCATOR which represents a LOB where the data of a JSON attribute may be stored. You can use this LOB\_RESULT\_LOCATOR with LOB FNC routines to write the data to the JSON attribute. You must use FNC\_GetStructuredResultLobAttribute if the JSON data to be written is larger than 64000 bytes. If the JSON data is equal to or less than 64000 bytes, you can still use FNC\_GetStructuredResultLobAttribute to write the data; however, you may get better performance if you use FNC\_SetStructuredAttribute instead.

By default, a structured UDT that an external routine returns has all of its attributes (except attributes that are themselves structured UDTs) set to null. If an external routine does not append any data to a LOB attribute of a structured type, that attribute remains null.

Setting the result indicator argument to -1 for an external routine that uses parameter style SQL discards any data that was appended to a LOB attribute. An external routine that appends data to a LOB attribute cannot reset the LOB attribute back to null if the result indicator argument is not set to -1.

A LOB attribute of a structured type that is defined to be an INOUT parameter to an external stored procedure retains the input version if no data is appended to the LOB attribute.

An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

### Example Using FNC\_GetStructuredResultLobAttribute

```
void document_t_lowerCase( UDT_HANDLE *documentUdt,
                          UDT_HANDLE *resultDocumentUdt,
                          char         sqlstate[6])
{
    LOB_RESULT_LOCATOR resultDocLoc;

    /* Get a LOB_RESULT_LOCATOR for the result doc attribute. */
    FNC_GetStructuredResultLobAttribute(*resultDocumentUdt, "doc",
                                       &resultDocLoc);
}
```

## FNC\_GetStructuredResultLobAttributeByNdx

Returns the locator of a LOB attribute of a structured type that is defined to be the return value of a UDF or UDM or an INOUT or OUT parameter to an external stored procedure.

### Syntax

```
void
FNC_GetStructuredResultLobAttributeByNdx ( UDT_HANDLE udtHandle,
                                           int *attributeIndex,
                                           LOB_RESULT_LOCATOR *object )
```

### Syntax Elements

#### *udtHandle*

the handle to a structured UDT that is defined to be one of the following:

- The return value of a UDF or UDM or an INOUT or OUT parameter to an external stored procedure
- A nested attribute in the structured UDT return value of a UDF or UDM or an INOUT or OUT parameter to an external stored procedure

#### *attributeIndex*

the index of a non-nested LOB attribute within the UDT.

The range of values is from 0 to *i*-1, where 0 is the index of the first attribute in the UDT and *i* is the number of top-most level attributes in the UDT.

For example, consider a structured UDT called `passportUDT` that has two attributes: the first attribute is an `INTEGER` attribute called `ID` and the second attribute is a `LOB` attribute called `photo`. To get the locator of the `photo` attribute, use an index of 1.

***object***

a pointer to the `LOB` locator for the `LOB` attribute.

## Usage Notes

You can use `FNC_GetStructuredResultLobAttributeByNdx` to obtain the `LOB` locator for a distinct type attribute that represents a `LOB` type.

After you obtain the `LOB` locator for the attribute of the structured type, use the `LOB` access functions, such as `FNC_LobOpen`, to append data.

By default, a structured UDT that an external routine returns has all of its attributes (except attributes that are themselves structured UDTs) set to null. If an external routine does not append any data to a `LOB` attribute of a structured type, that attribute remains null.

Setting the result indicator argument to -1 for an external routine that uses parameter style `SQL` discards any data that was appended to a `LOB` attribute. An external routine that appends data to a `LOB` attribute cannot reset the `LOB` attribute back to null if the result indicator argument is not set to -1.

A `LOB` attribute of a structured type that is defined to be an `INOUT` parameter to an external stored procedure retains the input version if no data is appended to the `LOB` attribute.

An external stored procedure that uses `CLlv2` to execute `SQL` must wait for any outstanding `CLlv2` requests to complete before calling this function.

To get the locator of a nested `LOB` attribute in a structured UDT that is defined to be the return value of a UDF or UDM, or `OUT` or `INOUT` parameter to an external stored procedure, follow these steps:

1. Call `FNC_GetStructuredAttributeByNdx` to obtain the UDT handle of the next structured UDT attribute in the path to the target attribute.

Repeat this step, passing in the newly obtained UDT handle as the *udtHandle* argument, *n-2* times, where *n* is the nesting level of the target attribute.

2. Call `FNC_GetStructuredResultLobAttributeByNdx`, passing in the UDT handle of the structured UDT attribute that contains the target `LOB` attribute.

## Example Using `FNC_GetStructuredResultLobAttributeByNdx`

```
void document_t_lowerCase( UDT_HANDLE *documentUdt,
                          UDT_HANDLE *resultDocumentUdt,
                          char         sqlstate[6])
{
    LOB_RESULT_LOCATOR resultDocLoc;

    /* Get a LOB_RESULT_LOCATOR for the result doc attribute. */
}
```

```
FNC_GetStructuredResultLobAttributeByNdx(*resultDocumentUdt,
0,                                     &resultDocLoc);
```

## FNC\_GetUDTHandles

Returns one or more UDT handles that can be used to operate on the elements of an ARRAY data type whose element type is UDT.

### Syntax

```
void
FNC_GetUDTHandles ( ARRAY_HANDLE  aryHandle,
                    int           numHandles,
                    void          *returnValue,
                    long          bufSize )
```

### Syntax Elements

#### *aryHandle*

the handle to an ARRAY type that is defined to be an input parameter to an external routine.

#### *numHandles*

the number of UDT handles that you want to retrieve.

#### *returnValue*

a pointer to a buffer that FNC\_GetUDTHandles uses to return the UDT handles. You must allocate 4 bytes for each UDT handle you want to retrieve.

#### *bufSize*

the size in bytes that was allocated for the *returnValue* buffer.

### Usage Notes

FNC\_GetUDTHandles takes *aryHandle*, *numHandles*, and *bufSize* as input arguments and returns the UDT handles requested in the *returnValue* buffer.

The ARRAY specified by *aryHandle* must have an element type of UDT. Before calling FNC\_GetUDTHandles, allocate the *returnValue* buffer with enough space to hold the desired number of UDT handles. You must allocate 4 bytes for each UDT handle you want to retrieve, that is  $4 * \text{numHandles}$ . Be sure to release allocated resources after you process the data.

To access and set the value of a distinct UDT or attribute values of a structured UDT, see [UDT Interface](#).

You can call `FNC_GetUDTHandles` to retrieve one or more UDT handles, operate on the set of UDTs, then call `FNC_SetArrayElements` or `FNC_SetArrayElementsWithMultiValues` with these UDTs, without being required to call `FNC_GetArrayElements`.

No more than 2000 UDT handles may be allocated during the execution of any UDF, UDM, or external stored procedure. Therefore, no more than 2000 UDT handles may be retrieved in any call to `FNC_GetUDTHandles`, and if other FNC calls in your routine also allocate UDT handles, this limit will be even lower.

### Example Using `FNC_GetUDTHandles`

This example is based on the following ARRAY definition:

```
/*Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE phonenumbers_ary AS VARRAY(5) OF CHAR(10);
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];

/* This function sets the values of a set of consecutive elements */
/* which are UDTs by first obtaining a set of UDT handles, */
/* performing operations on them, and then calling */
/* FNC_SetArrayElementsWithMultiValues. */

void setMultiElement( ARRAY_HANDLE *ary,
                      INTEGER      *result,
                      char          sqlstate[6])
{
    int numHandles;
    void *newValues;
    long newValuesBufSize;
    bounds_t *arrayInterval;
    NullBitVecType *NullBitVector;
    long NullBitVectorBufSize;

    numHandles = 3;
    newValuesBufSize = sizeof(UDT_HANDLE)*numHandles;
    newValues = (void*)FNC_malloc(newValuesBufSize);
    FNC_GetUDTHandles(*ary, numHandles, newValues, newValuesBufSize);

    /* Perform some operations on the UDTs using existing FNC routines.*/

    ...

    /* Set the values of arrayInterval to correspond to the range [1:3]*/
    arrayInterval = (bounds_t*)FNC_malloc(sizeof(bounds_t));
    arrayInterval[0].lowerBound = 1;
```



```

arrayInterval[0].upperBound = 3;

/* Allocate space for NullBitVector and set all bits to Present. */
numHandles % 8 == 0 ? NullBitVectorBufSize = numHandles / 8 :
    NullBitVectorBufSize = numHandles / 8 + 1;

NullBitVector = (NullBitVecType*)FNC_malloc(NullBitVectorBufSize);
FNC_SetNullBitVector(NullBitVector, -1, 1, NullBitVectorBufSize);

/* Set the values of the UDTs operated on above. */
FNC_SetArrayElementsWithMultiValues(ary, arrayInterval, newValues,
    newValuesBufSize, NullBitVector, NullBitVectorBufSize);

FNC_free(NullBitVector);
FNC_free(arrayInterval);
FNC_free(newValues);
}

```

## FNC\_GetVarCharLength

Returns the length, in bytes, for an external routine input argument that has a VARCHAR data type.

The length returned is the length of the VARCHAR data when the external routine is invoked.

If the server character set of the VARCHAR data is UNICODE, the value returned is the number of characters multiplied by two.

## Syntax

```

int
FNC_GetVarCharLength(void *inputString);

```

### Syntax Elements

#### *inputString*

a pointer to an external routine input argument that has a data type of VARCHAR.

## Usage Notes

You can use *FNC\_GetVarCharLength* instead of *strlen* to get the length of the input string.

Undefined results occur if the *inputString* argument points to data that is not VARCHAR.

## Example Using FNC\_GetVarCharLength

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"

void strrev13(VARCHAR_LATIN *input,
              int           inplen,
              VARBYTE       *result)
{
    int i;
    for (i=0; i< inplen; i++)
        result->bytes[inplen-i-1] = input[i];

    result->length = inplen;
/*
    memcpy(result,input,inplen);
*/
}

void strrevcomp_varchar(VARCHAR_LATIN *InputValue,
                        VARBYTE       *ResultValue,
                        char           sqlstate[6])
{
    int nullIndicator;
    int length = FNC_GetVarCharLength(InputValue);

    if (InputValue == NULL)
    {
        strcpy(sqlstate, "03288");
        return;
    }

    strrev13(InputValue, length, ResultValue);
    strcpy(sqlstate, "00000");
}
```

## FNC\_GetXML

Gets the value of an XML type.

## Syntax

```
void
FNC_GetXML(XML_HANDLE      xmlHandle,
           unsigned char *xmlBuffer,
           int             xmlBufferSize,
           int             xmlSize)
```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be an input parameter to an external routine.

### *xmlBuffer*

A pointer to the buffer that will hold the XML string.

### *xmlBufferSize*

The size in bytes of the *xmlBuffer* passed to FNC\_GetXML.

### *xmlSize*

The size in bytes of the XML string returned by FNC\_GetXML.

## Usage Notes

FNC\_GetXML is used to get the value of an XML type. The value will be in the UNICODE character set. The XML\_HANDLE for the XML type is passed in as input along with a pointer to a buffer (*xmlBuffer*) and the size of that buffer (*xmlBufferSize*). *xmlBufferSize* must be large enough to hold the XML value.

Note that the XML value can contain XML data that is not well-formed.

FNC\_GetXML can only be used for inline XML values, that is XML values less than 64K in size. You can use FNC\_GetXMLInfo to determine the maximum length of the XML value. You will get an error if you use FNC\_GetXML with a LOB-based XML value.

## Example

See [Example: FNC\\_GetXMLInfo, FNC\\_GetXML, and FNC\\_SetXML](#).

## FNC\_GetXMLBlob

Returns a LOB locator for the BLOB representation of the XML type.

## Syntax

```
void
FNC_GetXMLBlob(XML_HANDLE    xmlHandle,
                LOB_LOCATOR  *xmlBlob)
```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be an input parameter to an external routine.

### *xmlBlob*

A LOB locator for the BLOB representation of the XML type.

## Usage Notes

FNC\_GetXMLBlob is used to get a LOB locator for the BLOB representation of the XML type. The BLOB will be in the UTF-8 encoding.

The XML handle *xmlHandle* is passed as input. The LOB locator value *xmlBlob* is returned from the function. You can then use LOB FNC routines to read from the *xmlBlob* locator.

Note that the BLOB representation of the XML can contain XML data that is not well-formed.

FNC\_GetXMLBlob can only be used with LOB-based XML values, otherwise you will get an error.

## Example: Using FNC\_GetXMLBlob to Retrieve an XML Type Input Value

The following function takes an XML type parameter and retrieves the BLOB representation of the XML. It returns the BLOB back to the user.

```
CREATE FUNCTION xmlUDF5(P1 XML)
RETURNS BLOB AS LOCATOR
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF5!xmlUDF5.c';

void xmlUDF5(XML_HANDLE *xml_handle,
             LOB_RESULT_LOCATOR *return_blob,
             int* indicator_thisXML,
             int* indicator_returnLOB,
             char sqlstate[6],
```

```

        SQL_TEXT    extname[129],
        SQL_TEXT    specific_name[129],
        SQL_TEXT    error_message[257] )
{
    LOB_LOCATOR xmlBlob;
    BYTE buffer[100000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_XMLSize_t xmlSize;
    int numLobs;

    /* Get the XML Size */
    FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);

    if(numLobs ==1 )
    {

        /* Read XML value */
        FNC_GetXMLBlob(*xml_handle,&xmlClob);

        /* Use LOB FNC calls to read the XML BLOB and return it
*/

        FNC_LobOpen(*xmlBlob, &id, 0, 0);
        while( FNC_LobRead(id, buffer, 100000, &actlen) == 0 && !trunc_err)
            trunc_err = FNC_LobAppend(*return_blob, buffer, actlen, &actlen);

        FNC_LobClose(id);
        *indicator_returnLob = 0;
    }
}

```

## FNC\_GetXMLByte

Returns an XML value into a buffer in UTF-8 encoding.

### Syntax

```

void
FNC_GetXMLByte(XML_HANDLE    xmlHandle,
               byte          *xmlBuffer,
```

```

        int          xmlBufferSize,
        int          *xmlSize)

```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be an input parameter to an external routine.

### *xmlBuffer*

A pointer to the buffer that will hold the XML string.

### *xmlBufferSize*

The size in bytes of the *xmlBuffer* passed to FNC\_GetXMLByte.

### *xmlSize*

The size in bytes of the XML string returned by FNC\_GetXMLByte.

## Usage Notes

FNC\_GetXMLByte is used to get the value of an XML type in UTF-8 binary encoding.

The XML\_HANDLE is passed in as input along with a pointer to a buffer (*xmlBuffer*) and the buffer size (*xmlBufferSize*). *xmlBufferSize* must be large enough to hold the XML value.

Note that the XML value can contain XML data that is not well-formed.

FNC\_GetXMLByte can only be used for inline XML values, that is XML values less than 64K. You can use FNC\_GetXMLInfo to determine the maximum length of the XML value. If FNC\_GetXMLByte is used with a LOB-based XML value, you will get an error.

## Example

See [Example: FNC\\_GetXMLByte and FNC\\_SetXMLByte](#).

## FNC\_GetXMLClob

Returns a LOB locator for the CLOB representation of the XML type.

## Syntax

```

void
FNC_GetXMLClob(XML_HANDLE    xmlHandle,
               LOB_LOCATOR *xmlClob)

```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be an input parameter to an external routine.

### *xmlClob*

The LOB locator for the CLOB representation of the XML type.

## Usage Notes

FNC\_GetXMLClob is used to get a LOB locator for the CLOB representation of the XML type. The CLOB will be in the UNICODE character set.

The XML handle *xmlHandle* is passed as input. The LOB locator value *xmlClob* is returned from the function. The user can then use LOB FNC routines to read from the *xmlClob* locator.

Note that the CLOB representation of the XML can contain XML data that is not well-formed.

FNC\_GetXMLClob can only be used with LOB-based XML values, otherwise you will get an error.

## Example: Using FNC\_GetXMLClob to Retrieve an XML Type Input Value

The following function takes an XML type parameter and retrieves the CLOB representation of the XML. It returns the CLOB back to the user.

```
CREATE FUNCTION xmlUDF2(P1 XML)
RETURNS CLOB AS LOCATOR
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF2!xmlUDF2.c';

void xmlUDF2(XML_HANDLE *xml_handle,
             LOB_RESULT_LOCATOR *return_clob,
             int* indicator_thisXML,
             int* indicator_returnLOB,
             char sqlstate[6],
             SQL_TEXT extname[129],
             SQL_TEXT specific_name[129],
             SQL_TEXT error_message[257] )
{
    LOB_LOCATOR xmlClob;
    BYTE buffer[100000];
```

```

LOB_CONTEXT_ID id;
FNC_LobLength_t actlen;
int trunc_err = 0;
int numLobs;
FNC_XMLSize_t xmlSize;

FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);

    if(numLobs == 1)
    {

        /* Read XML value */
        FNC_GetXMLClob(*xml_handle,&xmlClob);

        /* Use LOB FNC calls to read the XML CLOB and return it
        */

        FNC_LobOpen(*xmlClob, &id, 0, 0);
        while( FNC_LobRead(id, buffer, 100000, &actlen) == 0 && !trunc_err)
            trunc_err = FNC_LobAppend(*return_clob, buffer, actlen, &actlen);

        FNC_LobClose(id);
        *indicator_returnLob = 0;
    }
}

```

## FNC\_GetXMLInfo

Returns information about the XML value including its size and whether it stores its value as a LOB.

### Syntax

```

void
FNC_GetXMLInfo(XML_HANDLE      xmlHandle,
               FNC_XMLSize_t *maxLength,
               int              numLobs)

```

### Syntax Elements

#### *xmlHandle*

A handle to an XML type that is defined to be an input parameter to an external routine.



***maxLength***

The maximum length possible for this XML value.

***numLobs***

If this XML object stores its value as a LOB, *numLobs* is 1, otherwise *numLobs* is 0.

**Usage Notes**

FNC\_GetXMLInfo is used to return information about the XML value such as its maximum size and whether it uses a LOB to store its value. The XML\_HANDLE for the XML value is passed in as input and the maximum size and whether it can store its value as a LOB is returned to the caller.

**Example: FNC\_GetXMLInfo, FNC\_GetXML, and FNC\_SetXML**

This function takes an XML parameter and retrieves the XML value. It returns the string back in the XML return parameter.

```
CREATE FUNCTION xmlUDF1(P1 XML)
RETURNS XML
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF1!xmlUDF1.c';

void xmlUDF1(XML_HANDLE* xml_handle,
             XML_HANDLE* return_handle,
             int*        indicator_thisXML,
             int*        indicator_returnXML,
             char         sqlstate[6],
             SQL_TEXT    extname[129],
             SQL_TEXT    specific_name[129],
             SQL_TEXT    error_message[257] )
{
    FNC_XMLSize_t xmlSize;
    char* xmlBuffer;

    int xmlBufferSize;
    int numLobs;

    /* Get the XML Size. Note we use size + 2 to account for
       the Unicode null termination character. */
    FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);
```

```

    if(numLobs == 0)
    {
        xmlBuffer = (char*)FNC_Malloc(xmlSize + 2);
        xmlBufferSize = xmlSize + 1;

        /* Get the XML Value */
        FNC_GetXML(*xml_handle, xmlBuffer, xmlBufferSize, &xmlSize);

        /* Set the XML string as return value */
        FNC_SetXML(*return_handle, xmlBuffer, xmlSize);

        *indicator_returnXML = 0;
        FNC_free(xmlBuffer);
    }
}

```

## FNC\_GetXMLResultBlob

Get a result LOB locator that will be used to set the XML return value.

### Syntax

```

void
FNC_GetXMLResultBlob(XML_HANDLE      xmlHandle,
LOB_RESULT_LOCATOR* xmlBlob)

```

### Syntax Elements

#### *xmlHandle*

A handle to an XML type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *xmlBlob*

A pointer to the LOB locator that is used to set the XML type return value.

### Usage Notes

FNC\_GetXMLResultBlob is used to get a result LOB locator that will be used to set the XML return value.

The XML handle *xmlHandle* is passed as input and a LOB locator *xmlBlob* is returned from the function. You can then use LOB FNC routines to set the BLOB value using the *xmlBlob* locator. UTF-8 encoding should be used to write to the BLOB.

FNC\_GetXMLResultBlob can only be used with LOB-based XML return values, otherwise you will get an error.

### Example

See [Example: FNC\\_GetXMLResultBlob and FNC\\_SetXMLBlob](#).

## FNC\_GetXMLResultClob

Returns a result LOB locator that will be used to set the XML return value.

### Syntax

```
void
FNC_GetXMLResultClob(XML_HANDLE      xmlHandle,
                     LOB_RESULT_LOCATOR *xmlClob)
```

### Syntax Elements

#### *xmlHandle*

A handle to an XML type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *xmlClob*

A pointer to the LOB locator that is used to set the XML type return value.

### Usage Notes

FNC\_GetXMLResultClob is used to get a result LOB locator that will be used to set the XML return value.

The XML handle *xmlHandle* is passed as input and a LOB locator *xmlClob* is returned from the function. You can then use LOB FNC routines to set the CLOB value using the *xmlClob* locator. You must use the UNICODE character set to write to the CLOB.

FNC\_GetXMLResultClob can only be used with LOB-based XML return values, otherwise you will get an error.

### Example

See [Example: FNC\\_GetXMLResultClob and FNC\\_SetXMLClob](#).

## FNC\_GLOP\_Global\_Copy

Copies GLOP data in one mapping instance to all mapping instances on all vprocs on all nodes. You can copy a specific range of data.

The modification attribute of the GLOP data must be set to globally modifiable. (You set this attribute when you call the `DBCExtension.GLOP_Add` stored procedure to add GLOP data and GLOP mappings to a GLOP set.)

`FNC_GLOP_Global_Copy` returns an integer that indicates success or failure.

Value	Meaning
0	Data was copied.
-1	The index is not valid or the map does not exist.
-2	The copy attempt timed out.
-3	Cannot copy read-only or read/write GLOP.
-4	Address range not valid.

## Syntax

```
int FNC_GLOP_Global_Copy( int    Index,
                          void *Data_Addr,
                          void *GLOP_Start_Addr,
                          int    Length)
```

### Syntax Elements

#### *Index*

the index to the GLOP to where the data is to be copied.

The `GLOP_mode` field in the `GLOP_ref_t` structure that was returned by a previous call to `FNC_Get_GLOP_Map` must be set to `GLOP_M` (3) or `GLOP_SM` (5).

#### *Data\_Addr*

the starting address of the data to be copied. This could be from any valid memory location the external routine has access to. It may be an address inside the GLOP mapping. A nondestructive copy is performed if `Data_Addr` and `GLOP_Start_Addr` overlap or are the same.

#### *GLOP\_Start\_Addr*

the starting address of where the data is copied to inside the GLOP mapping.

#### *Length*

length in bytes of the data to be copied. The exact number of bytes specified will be copied.

## Usage Notes

The function first attempts to obtain a local write lock on each vproc. If any one of the vprocs cannot get the lock, the function returns a value of -2 and releases all locks. The call can be tried again if desired. After the lock is obtained on all vprocs, the desired range of memory from the caller is sent to all vprocs. The lock is not released until all copies are updated. A successful call returns 0 for the result. Here are some additional considerations:

- This is an expensive operation. Avoid frequent calls to this function.
- Make sure that the appropriate external routine performs the global updates and that they do not step over each other.
- The best practice is to use some type of versioning information in the GLOP data to make sure that the external routine is using the correct version. This must be coordinated with the external routine.
- Global copy is not designed to redistribute a lot of data. Do not try to use it to duplicate rows. Instead, use SQL statements to do that. Using it in this manner compromises system performance.
- The global copy changes the designated range for all data on all vprocs (all nodes) for the given GLOP data in the given set.
- The global copy only makes changes to all existing GLOP data on all vprocs associated with the type and context. If additional contexts are established subsequent to this call, they will have the initial GLOP context. There is no attempt to keep any future GLOP data in sync with the changes made by the global copy command. If the GLOP data is transaction based, then it only updates those GLOP mappings that are associated with the transaction. If the data is user based, then it updates all GLOP mappings associated with the user that are currently mapped at the time this command is executed.

## Related Information

For more information on GLOP data, see [Global and Persistent Data](#)

## FNC\_GLOP\_Lock

Lock a vproc-local, read/write GLOP for reading or writing.

FNC\_GLOP\_Lock returns an integer that indicates success or failure.

Value	Meaning
0	Lock was granted.
-1	The index is not valid or the map does not exist.
-2	The lock attempt timed out.
-3	Cannot lock a read-only GLOP.
-4	The GLOP is already locked by this routine. The routine may only perform one lock of any kind at a time.

## Syntax

```
int FNC_GLOP_Lock( int      Index,
                  LockType_en Lock)
```

### Syntax Elements

#### *Index*

the index to the GLOP to be locked.

The valid range of values is 0 to 7, indicating an entry in the GLOP map structure.

#### *Lock*

the type of lock to apply, where valid values are:

- FNC\_READ\_LOCK
- FNC\_WRITE\_LOCK

## Usage Notes

Use this function for a read/write GLOP that is shared by more than one external routine at the same time. Because it is not possible to determine whether a GLOP is being shared, the best practice is to use locks at all times. The only time not to use them is when the design of the data does not require a lock. For example, a design that divides the data in such a manner that ensures that all external routines access their own specific address range such that there is no possibility that another external routine can use the data at the same time can get by without using locks. (This scenario is highly unlikely.) For example, if the GLOP data gets updated in memory specifically at 12 AM everyday, all external routines can ignore placing read locks if the time is not close to 12 AM. In any case, any usage design needs to consider this issue.

Multiple read locks are granted and only one write lock is granted. All lock requests have a 130 second timeout associated with them. If the lock request cannot be granted within 130 seconds the function returns a value of -2. The caller must check for that and reapply the request or move on to do something else. The reason for doing this is to avoid any permanent deadlock. The external routine can always decide to abort or try again. Also, any granted lock has a timeout of 2 minutes. This default can be changed with a configuration parameter. If the GLOP is locked for more than two minutes, the system aborts the transaction. If the external routine terminates with a lock in place, the system aborts the transaction. This is done to avoid external routine code from locking the memory for extended time periods. It avoids having the database add complex code to try and solve global deadlock issues. Having the timeout also encourages efficient usage of the GLOP while locked. The lock command is only relevant to read/write data. There is no point in locking read-only data and it is not allowed.

This is an honor system. The external routine can change memory (if it is read/write) at any time with no locks. It can also read the memory with no locks. However the external routine must know whether doing that makes sense and could cause problems for other external routines.

Do not assume that because a read/write GLOP is mapped for example at the request level that locks are not needed, because that is not necessarily the case when there are parallel steps in the request. It still requires a lock to avoid any conflict.

## Related Information

For more information on GLOP data, see [Global and Persistent Data](#)

## FNC\_GLOP\_Map\_Page

Map different pages of a read-only GLOP or reload the single page of a read/write or globally modifiable GLOP.

FNC\_GLOP\_Map\_Page returns an integer that indicates success or failure.

Value	Meaning
0	Page was mapped.
-1	The index is not valid or the map does not exist.
-2	Invalid page number.
-3	Lock conflict. The GLOP is locked by the caller and the system cannot change the page.

## Syntax

```
int FNC_GLOP_Map_Page( int Map_Index,
                      int Page)
```

### Syntax Elements

#### *Map\_Index*

the index to the GLOP entry being referred to in the GLOP map.

The valid range of values is 0 to 7, indicating an entry in the GLOP map structure.

#### *Page*

the number of the page to map.

## Usage Notes

Read-only GLOP data can be split into a number of pages, each of which can be overlaid with the current GLOP data in memory. The page to initially map for read-only GLOP data can be specified when the GLOP is added or changed. Read/write or globally modifiable GLOP data can only consist of one page.

Remapping page one of such a GLOP will remap the original data contents of the GLOP. Any changed data in the read/write page is thrown away.

All read-only pages are the same size and depend on the GLOP\_Length field of the GLOP set row.

The GLOP read/write data must not be locked when this call is made. It places an internal exclusive lock that blocks all read and write locks while a page is being replaced.

Rules to be aware of:

- You can map the same type of GLOP data in multiple map indexes for read-only GLOP data. That way you can map different pages of the same GLOP data at the same time. This has to be set up that way in the GLOP\_Map table. This cannot be done for 'RO', 'US' or 'XR' GLOPs.
- When reloading page one map for a read write GLOP, an automatic write lock is placed to prevent any external routine from reading or writing the GLOP while the data is being reinitialized with the original contents. If you use this as part of the design, make sure that all external routines are aware that this could happen.
- Different pages for read-only GLOP data can be mapped by different external routines at the same time.

## Example Using FNC\_GLOP\_Map\_Page

Consider an external routine that maps one of five XML syntax rules pages depending on the type of document it must process. The XML rules are kept in a read-only system map in the "XML\_Document\_Syntax" set. The reason for using a system map is that the GLOP is read-only, so only one copy needs to exist and the XML document syntax rules are system wide and all documents need to use the same syntax.

Remember that more than one system type GLOP can be defined in a set. So if there is a need for specific generic system GLOP data, it can be mapped also.

```
#define XML_INDEX 0
XML_Rules    Current_XML_Rules;
GLOP_Map_t *MyGLOP;
int          Page_map;
int          glop_status

. . .

glop_status = FNC_Get_GLOP_Map(&MyGLOP);
if (!glop_status)
    ... process error: Not a member of any GLOP table
if ( MyGLOP->GLOP[XML_INDEX].GLOP_Ptr == NULL)
{
    ... process error: XML Syntax map does not exist
};
```



```

/* Determine page to map, based on "document_type" input parameter */
switch (*document_type)
{
    case WEB_page:
        Page_map = 1;
        break;
    case PDF_page:
        Page_map = 2;
        break;
    case Text_page:
        Page_map = 3;
        break;
    case Graphic_page:
        Page_map = 4;
        break;
    case Formula_page:
        Page_map = 5;
        break;
    Default:
        /* error: invalid documents option - generate an error */
}

if (MyGLOP->GLOP[XML_INDEX].page != Page_map)
{
    if (!FNC_GLOP_Map_Page(XML_INDEX, Page_map))
        ... error: could not map page
}

Current_XML_Rules = MyGLOP->GLOP[XML_INDEX].GLOP_ptr;

/* process the document */

```

## Related Information

For more information on GLOP data, see [Global and Persistent Data](#)

## FNC\_GLOP\_Unlock

Unlock a read/write GLOP that was previously locked with a call to FNC\_GLOP\_Lock.

FNC\_GLOP\_Unlock returns an integer with the following possible values.

Value	Meaning
0	Unlock was successful.

Value	Meaning
-1	The index is not valid or the entry is not mapped.
-2	There was no lock.

## Syntax

```
int FNC_GLOP_Unlock( int    Index)
```

### Syntax Elements

#### *Index*

the index to the GLOP to be unlocked.

The valid range of values is 0 to 7, indicating an entry in the GLOP map structure.

## Usage Notes

If some other external routine was waiting for the lock, it might or might not get control before the function call returns.

## Related Information

For more information on GLOP data, see [Global and Persistent Data](#)

## FNC\_LobAppend

Appends a sequence of bytes to a large object that is contained in a UDT or defined to be the result of a UDF, UDM, or external stored procedure, or defined to be the column of a table operator output stream.

IF the ...	THEN FNC_LobAppend returns ...
operation is successful	0 The <i>actual_length</i> argument is set to the value of <i>data_length</i> .
truncation occurs	-1 The <i>actual_length</i> argument is set to the number of bytes actually appended.

## Syntax

```
int  
FNC_LobAppend ( LOB_RESULT_LOCATOR    object,
```

```

        BYTE                *data,
        FNC_LobLength_t     data_length,
        FNC_LobLength_t     *actual_length )

```

## Syntax Elements

### *object*

the object.

The *object* argument must be contained in a UDT or the *result* parameter of the UDF, UDM, or external stored procedure, or the result of calling FNC\_LobCol2Loc on a LOB column in an output stream.

### *data*

a pointer to the start of the data to be appended to *object*.

### *data\_length*

the number of bytes to be appended. If *data\_length* is 0, then the function has no effect.

### *actual\_length*

a pointer to a variable that FNC\_LobAppend sets to the number of bytes actually appended. This can be less than *data\_length* if truncation occurs.

## Usage Notes

A UDF, UDM, or external stored procedure can return an empty LOB value by returning without calling FNC\_LobAppend.

The arguments are validated to the extent possible. If an argument is not valid, the request that invoked the UDF, UDM, or external stored procedure fails and typically returns an error that looks like this:

```

7554 Invalid LOCATOR argument to LOB access function in
UDF      database_name.udf_name.

```

Control does not return to the UDF, UDM, or external stored procedure.

FNC\_LobAppend needs to be called to populate a LOB that is defined as a column in an output stream of a table operator. Before calling FNC\_LobAppend, you must call FNC\_LobCol2Loc to obtain the LOB\_RESULT\_LOCATOR passed as the first argument to FNC\_LobAppend.

## Restrictions

An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

## Example Using FNC\_LobAppend

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#define BUFFER_SIZE 100000

void concat2( LOB_LOCATOR *a,
              LOB_LOCATOR *b,
              LOB_RESULT_LOCATOR *result,
              char sqlstate[6] )
{
    FNC_LobLength_t actlen;
    BYTE            buffer[BUFFER_SIZE];
    LOB_CONTEXT_ID  id;
    int             trunc_err = 0;

    /* Append the first argument. */

    FNC_LobOpen(*a, &id, 0, 0);

    while( FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen) == 0
          && !trunc_err )
    trunc_err = FNC_LobAppend(*result, buffer, actlen, &actlen);

    FNC_LobClose(id);

    ...
}
```

## FNC\_LobClose

Releases all resources associated with a read context and reduces the number of outstanding read contexts by one, regardless of whether the end of data was reached.

## Syntax

```
void
FNC_LobClose ( LOB_CONTEXT_ID  ctxid )
```

### Syntax Elements

***ctxid***

the context identifier that was previously returned by the FNC\_LobOpen function.

## Usage Notes

Use FNC\_LobClose when the maximum number of outstanding read contexts has been reached and a UDF, UDM, or external stored procedure must access another large object.

You can use FNC\_LobClose regardless of whether the end of data was previously reached. However, for best performance when a UDF, UDM, or external stored procedure does not intend to read an object all the way to the end, use the *maxlength* argument of FNC\_LobOpen to access a subset of the object.

When a UDF, UDM, or external stored procedure returns normally, Vantage implicitly closes all outstanding read contexts.

The LOB\_CONTEXT\_ID argument is validated to the extent possible. If the argument is not valid, the request that invoked the UDF, UDM, or external stored procedure fails and typically returns an error that looks like this:

```
7554 Invalid CONTEXT_ID argument to LOB access function in
UDF  database_name.udf_name.
```

Control does not return to the UDF, UDM, or external stored procedure.

### Restrictions

An external stored procedure that uses CLlv2 to execute SQL must wait for any outstanding CLlv2 requests to complete before calling this function.

## Example Using FNC\_LobClose

```
#define BUFFER_SIZE 500

void do_something ( LOB_LOCATOR      *a,
                   LOB_RESULT_LOCATOR *result,
                   char               sqlstate[6] )
```

```

{
    BYTE          buffer[BUFFER_SIZE];
    FNC_LobLength_t actlen;
    LOB_CONTEXT_ID id;

    FNC_LobOpen(*a, &id, 0, BUFFER_SIZE);
    FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen);
    FNC_LobClose(id);

    ...
}

```

## FNC\_LobCol2Loc

Converts an output stream column index into a LOB\_RESULT\_LOCATOR. This generated locator can then be used by FNC\_LobAppend. This function is used with table operators.

This function can only be called from a table operator.

## Syntax

```

LOB_RESULT_LOCATOR
FNC_LobCol2Loc (int  streamno,
                int  columnnr)

```

## Syntax Elements

### *streamno*

the output stream number. Currently only one output stream is supported; therefore, the only valid value is 0.

### *columnnr*

the column position of a LOB in the output stream. Valid values are 0..*n*-1 where *n* is the number of columns in the output stream. The type of the column in the output stream must be BLOB or CLOB.

## FNC\_LobLoc2Ref

Converts a locator into a persistent object reference.

## Syntax

```
void
FNC_LobLoc2Ref ( LOB_LOCATOR   locator,
                 LOB_REF       *ref )
```

### Syntax Elements

#### *locator*

the locator to convert to a persistent object reference.

#### *ref*

a pointer to the persistent object reference that FNC\_LobLoc2Ref creates.

## Usage Notes

Use the persistent object reference that FNC\_LobLoc2Ref creates to copy LOB values into the intermediate storage required for aggregate UDFs.

The arguments are validated to the extent possible. If an argument is not valid, the request that invoked the UDF fails and typically returns an error that looks like this:

```
7554 Invalid CONTEXT_ID LOCATOR argument to LOB access function in
UDF database_name.udf_name.
```

Control does not return to the UDF.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_LobLoc2Ref

```
#define BUFFER_SIZE 500

/* Aggregate intermediate record */
typedef struct
{
    FNC_LobLength_t length;
    LOB_REF         ref;
    BYTE            prefix[BUFFER_SIZE];
}
```

```

} intrec_t;
void Max_Blob(FNC_Phase      phase,
              FNC_Context_t  *fctx,
              LOB_LOCATOR    *x,
              LOB_RESULT_LOCATOR *result,
              int             *x_i,
              int             *result_i,
              char            sqlstate[6])
{
    intrec_t      *s1 = (intrec_t*) fctx->interim1;
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actsize;

    switch (phase)
    {
    case AGR_INIT:
        s1 = (intrec_t *)FNC_DefMem(sizeof(intrec_t));
        s1->length = FNC_GetLobLength(*x);
        FNC_LobOpen(*x, &id, 0, BUFFER_SIZE);
        FNC_LobRead(id, s1->prefix, BUFFER_SIZE, &actsize);
        FNC_LobClose(id);

        /* if the object length is less than the size
           allotted, pad the remaining bytes with zeros. */
        while (actsize < BUFFER_SIZE)
            s1->prefix[actsize++] = 0;
        FNC_LobLoc2Ref(*x, &s1->ref);

    case AGR_DETAIL:

        ...

        break;

    ...

    }
}

```

For a complete example that uses FNC\_LobLoc2Ref, see [C Aggregate Function Using LOBs](#).

## FNC\_LobOpen

Establishes a read context for subsequent sequential reads of a referenced object.



FNC\_LobOpen returns the length of the resulting data stream.

## Syntax

```
FNC_LobLength_t
FNC_LobOpen ( LOB_LOCATOR      object,
              LOB_CONTEXT_ID  *ctxid,
              FNC_LobLength_t start,
              FNC_LobLength_t maxlength )
```

### Syntax Elements

#### *object*

the object.

#### *ctxid*

a context identifier.

#### *start*

the starting byte offset, relative to 0, for the read operation. If this value is greater than or equal to the length of the object, then the resulting data stream will have a zero length.

#### *maxlength*

the maximum number of bytes to read. If this value is 0, or if  $start + maxlength$  is greater than the object length, then the read operation terminates when the length of the object is exhausted.

## Usage Notes

Before you call FNC\_LobRead to access the value of a large object, you must call FNC\_LobOpen to establish a read context.

For best performance of a UDF, UDM, or external stored procedure that is not designed to read an entire object, use the *start* and *maxlength* arguments to specify a subset of the object. If the object is on a remote AMP, it saves the cost of moving unneeded data.

Calling FNC\_LobOpen increases the number of outstanding read contexts by one.

IF the UDF or external stored procedure result type is ...	THEN the maximum number of outstanding read contexts is ...
a large object	3
not a large object	4

The LOB\_LOCATOR argument is validated to the extent possible. If the argument is not valid, the request that invoked the UDF, UDM, or external stored procedure fails and typically returns an error that looks like this:

```
7554 Invalid LOCATOR argument to LOB access function in UDF
database_name.udf_name.
```

Control does not return to the UDF, UDM, or external stored procedure.

## Restrictions

An external stored procedure that uses CLlv2 to execute SQL must wait for any outstanding CLlv2 requests to complete before calling this function.

## Example Using FNC\_LobOpen

```
#define BUFFER_SIZE 500

void do_something ( LOB_LOCATOR *a,
                   LOB_RESULT_LOCATOR *result,
                   char sqlstate[6] )
{
    BYTE          buffer[BUFFER_SIZE];
    FNC_LobLength_t actlen;
    LOB_CONTEXT_ID id;

    FNC_LobOpen(*a, &id, 0, BUFFER_SIZE);
    FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen);
    FNC_LobClose(id);

    ...
}
```

## FNC\_LobOpen\_CL

Establishes a read context for subsequent sequential reads of a referenced object. This function is used with table operators.

FNC\_LobOpen\_CL returns the length of the resulting data stream.

## Syntax

```
FNC_LobLength_t
FNC_LobOpen_CL (void    *locator,
                LOB_CONTEXT_ID *ctxid,
                FNC_LobLength_t start,
                FNC_LobLength_t maxlength)
```

### Syntax Elements

#### *locator*

a client locator.

#### *ctxid*

a context identifier.

#### *start*

the starting byte offset, relative to 0, for the read operation. If this value is greater than or equal to the length of the object, then the resulting data stream will have a zero length.

#### *maxlength*

the maximum number of bytes to read. If this value is 0, or if *start* + *maxlength* is greater than the object length, then the read operation terminates when the length of the object is exhausted.

## Usage Notes

This function is used with table operators and differs from FNC\_LobOpen in the data type of the first parameter.

Before you call FNC\_LobRead to access the value of a large object, you must call FNC\_LobOpen\_CL to establish a read context.

For best performance of a table operator that is not designed to read an entire object, use the *start* and *maxlength* arguments to specify a subset of the object. If the object is on a remote AMP, it saves the cost of moving unneeded data.

Calling FNC\_LobOpen\_CL increases the number of outstanding read contexts by one. The maximum number of outstanding read contexts is 4.

The *locator* argument is validated to the extent possible. If the argument is not valid, the request that invoked the table operator fails and returns an error.

## Example Using FNC\_LobOpen\_CL

The following code reads a LOB from the input stream and copies its content into the output stream.

```
LOB_RESULT_LOCATOR    lrl_a;
int                   streamno=0;
int                   i;
BYTE                  *locator;
int                   locatorLength;
int                   null_ind;
FNC_LobLength_t       lobLength;
LOB_CONTEXT_ID        lobid;
BYTE                  Buffer[BufferSize];
int                   length;
int                   truncerr;
FNC_TblOpHandle_t     *Handle, *OutHandle;
.....

// Get locator for LOB in column i of output stream streamno
lrl_a = FNC_LobCol2Loc(streamno, i);
// Get client locator for input LOB
FNC_TblOpGetAttributeByNdx(Handle, i, (void **) &locator,
&null_ind, &locatorLength);
if (null_ind != -1) {
    // Open input LOB
    FNC_LobOpen_CL(locator, &lobid, 0, 0);
    truncerr = 0;
    // Read input LOB and append it to output LOB
    while( FNC_LobRead(lobid, Buffer, BufferSize, &length) == 0 && !truncerr)
        truncerr=FNC_LobAppend (lrl_a, Buffer, length, &actlen);
    // Close input LOB
    FNC_LobClose(lobid);
}
else
    // Set output column to NULL
    FNC_TblOpBindAttributeByNdx(OutHandle, i, locator, null_ind, locatorLength);
```

## FNC\_LobRead

Performs a sequential read using a specified read context.

IF the ...	THEN FNC_LobRead returns ...
operation is successful	0
object length or the length specified in the previous call to FNC_Open was exhausted	-1 The <i>actual_length</i> argument is set to 0.

## Syntax

```
int
FNC_LobRead ( LOB_CONTEXT_ID    ctxid,
              BYTE               *buffer,
              FNC_LobLength_t    buffer_length,
              FNC_LobLength_t    *actual_length )
```

### Syntax Elements

#### *ctxid*

a context identifier established by a previous call to FNC\_LobOpen.

#### *buffer*

the start of a buffer to place the data.

#### *buffer\_length*

the maximum number of bytes to read and place in *buffer*.

#### *actual\_length*

a pointer to a variable where FNC\_LobRead stores the actual number of bytes that were read and placed in *buffer*.

## Usage Notes

The LOB\_CONTEXT\_ID argument is validated to the extent possible. If the argument is not valid, the request that invoked the UDF, UDM, or external stored procedure fails and typically returns an error that looks like this:

```
7554 Invalid CONTEXT_ID argument to LOB access function in UDF
database_name.udf_name.
```

Control does not return to the UDF, UDM, or external stored procedure.

## Restrictions

An external stored procedure that uses CLlv2 to execute SQL must wait for any outstanding CLlv2 requests to complete before calling this function.

## Example Using FNC\_LobRead

```
#define BUFFER_SIZE 500

void do_something ( LOB_LOCATOR *a,
                   LOB_RESULT_LOCATOR *result,
                   char sqlstate[6] )
{
    BYTE          buffer[BUFFER_SIZE];
    FNC_LobLength_t actlen;
    LOB_CONTEXT_ID id;

    FNC_LobOpen(*a, &id, 0, BUFFER_SIZE);
    FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen);
    FNC_LobClose(id);

    ...
}
```

## FNC\_LobRef2Loc

Converts a persistent object reference into a locator.

## Syntax

```
LOB_LOCATOR
FNC_LobRef2Loc ( const LOB_REF  *ref )
```

## Syntax Elements

*ref*

a pointer to an object reference that was previously created by FNC\_LobLoc2Ref.

## Usage Notes

Use FNC\_LobRef2Loc to copy values from the intermediate storage into the result during the final aggregation phase in an aggregate UDF.

The LOB\_REF argument is validated to the extent possible. If the argument is not valid, the request that invoked the UDF fails. Control does not return to the UDF.

## Restrictions

An external stored procedure that uses CLlv2 to execute SQL must wait for any outstanding CLlv2 requests to complete before calling this function.

## Example Using FNC\_LobRef2Loc

```
#define BUFFER_SIZE 10000

/* Aggregate intermediate record */
typedef struct
{
    FNC_LobLength_t length;
    LOB_REF          ref;
    BYTE             prefix[500];
} intrec_t;

void Max_Blob(FNC_Phase          phase,
              FNC_Context_t      *fctx,
              LOB_LOCATOR        *x,
              LOB_RESULT_LOCATOR *result,
              int                 *x_i,
              int                 *result_i,
              char                sqlstate[6])
{
    intrec_t      *s1 = (intrec_t*) fctx->interim1;
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;

    switch (phase)
    {
    case AGR_INIT:

        ...
```

```

case AGR_DETAIL:

    ...

    break;
case AGR_FINAL:
    FNC_LobOpen(*(FNC_LobRef2Loc(&s1->ref)), &id, 0, 0);
    FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen);
    FNC_LobAppend(*result, buffer, actlen, &actlen);
    FNC_LobClose(id);
    break;

    ...

}

```

For a complete example that uses FNC\_LobRef2Loc, see [C Aggregate Function Using LOBs](#).

## FNC\_MBBGetValue

Returns a buffer containing six double precision values representing the xmin, ymin, zmin, xmax, ymax, and zmax coordinates that define an MBB (minimum bounding box) type.

### Syntax

```

void
FNC_MBBGetValue(GEO_HANDLE      mbbHandle,
                double *        mbbBuffer,
                int *            mbbSize )

```

### Syntax Elements

#### *mbbHandle*

A handle to an MBB type that is defined to be an input parameter to an external routine.

#### *mbbBuffer*

A pointer to a buffer that will hold the six double precision values that define the MBB.

#### *mbbSize*

Size in bytes of the value returned in the *mbbBuffer* buffer.



## Usage Notes

FNC\_MBBGetValue retrieves the xmin, ymin, zmin, xmax, ymax, and zmax values of an MBB type. The values are returned in the *mbbBuffer* as six double precision values. The caller of the function needs to allocate sufficient memory for the *mbbBuffer* buffer to hold six double precision values.

### Example: Using FNC calls to get an MBB type value

The following function takes an MBB type parameter and returns an array of six double precision values representing the xmin, ymin, zmin, xmax, ymax, and zmax values that define the MBB.

```
CREATE TYPE MBB_ARRAY AS DOUBLE PRECISION ARRAY[6];
CREATE FUNCTION mbbUDF1(P1 MBB)
RETURNS MBB_ARRAY
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!mbbUDF1!mbbUDF1.c';

void mbbUDF1 (GEO_HANDLE* mbb_handle,
              ARRAY_HANDLE* resultArray,
              int* indicator_thismbb,
              int* indicator_Result,
              char      sqlstate[6],
              SQL_TEXT  extname[129],
              SQL_TEXT  specific_name[129],
              SQL_TEXT  error_message[257] )
{
    double* mbbBuffer;
    int returnSize;
    int mbbBufferSize;
    array_info_t arrayInfo;
    bounds_t arrayScope[FNC_MAXARRAYDIMENSIONS];
    NullBitVecType      *NullBitVector;

    mbbBufferSize = 6 * sizeof(double);
    mbbBuffer = (double*)FNC_Malloc(mbbBufferSize);
    /* Get the MBB value */
    FNC_MBBGetValue(*mbb_handle, mbbBuffer, &returnSize);

    /* get element type information for the ARRAY */
    FNC_GetArrayTypeInfo(*((ARRAY_HANDLE*)resultArray),
                        &arrayInfo,
                        arrayScope);
}
```

```

/* Allocate a new NullBitVector */
NullBitVector = (NullBitVecType*)FNC_malloc(1);

/* Set array elements within the range. */
FNC_SetArrayElementsWithMultiValues(
    *((ARRAY_HANDLE*)resultArray),
    arrayScope,
    (void*)mbbBuffer,
    mbbBufferSize, NullBitVector,1);

*indicator_Result = 0;
FNC_free(NullBitVector);
FNC_free(mbbBuffer);
}

```

## FNC\_MBBSetValue

Set the value of an MBB (minimum bounding box) type using a buffer of six double precision values. The values represent the xmin, ymin, zmin, xmax, ymax, and zmax coordinates that define the MBB.

### Syntax

```

void
FNC_MBBSetValue(GEO_HANDLE      mbbHandle,
                double *        mbbValue)

```

### Syntax Elements

#### *mbbHandle*

A handle to an MBB type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *mbbValue*

A pointer to a buffer containing the six double precision values.

### Usage Notes

FNC\_MBBSetValue sets the value of an MBB type. A buffer *mbbValue* containing six double precision values is provided as input to the routine along with the MBB handle.

### Example: Using FNC calls to set an MBB type value

The following function takes six double precision values as input, and returns an MBB value.

```

CREATE FUNCTION mbbUDF1(P0 DOUBLE PRECISION, P1 DOUBLE PRECISION, P2
DOUBLE PRECISION,
                        P3 DOUBLE PRECISION, P4 DOUBLE PRECISION, P5
DOUBLE PRECISION)
RETURNS MBB
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!mbbUDF1!mbbUDF1.c';

```

```

void mbbUDF1 (DOUBLE_PRECISION* p0,
              DOUBLE_PRECISION * p1,
              DOUBLE_PRECISION * p2,
              DOUBLE_PRECISION * p3,
              DOUBLE_PRECISION * p4,
              DOUBLE_PRECISION * p5,
              GEO_HANDLE* resultMBB,
              int* indicator_p0,
              int* indicator_p1,
              int* indicator_p2,
              int* indicator_p3,
              int* indicator_p4,
              int* indicator_p5,
              int* indicator_Result,
              char      sqlstate[6],
              SQL_TEXT  extname[129],
              SQL_TEXT  specific_name[129],
              SQL_TEXT  error_message[257] )
{
    double* mbbBuffer;
    int returnSize;
    int mbbBufferSize;

    mbbBufferSize = 6 * sizeof_DOUBLE_PRECISION;
    mbbBuffer = (double*)FNC_Malloc(mbbBufferSize);
    /* Set the double precision values in the buffer */
    mbbBuffer[0] = *p0;
    mbbBuffer[1] = *p1;
    mbbBuffer[2] = *p2;
    mbbBuffer[3] = *p3;
    mbbBuffer[4] = *p4;
    mbbBuffer[5] = *p5;

```

```

    /* Set the MBB value */
    FNC_MBBSetValue(*resultMBB, mbbBuffer);

    *indicator_Result = 0;
    FNC_free(mbbBuffer);
}

```

## FNC\_MBRGetValue

Returns a buffer containing four double precision values representing the xmin, ymin, xmax, and ymax coordinates that define an MBR (minimum bounding rectangle) type.

### Syntax

```

void
FNC_MBRGetValue(GEO_HANDLE    mbrHandle,
                double *      mbrBuffer,
                int *         mbrSize )

```

### Syntax Elements

#### *mbrHandle*

A handle to an MBR type that is defined to be an input parameter to an external routine.

#### *mbrBuffer*

A pointer to a buffer that will hold the four double precision values that define the MBR.

#### *mbrSize*

Size in bytes of the value returned in the *mbrBuffer* buffer.

### Usage Notes

FNC\_MBRGetValue retrieves the xmin, ymin, xmax, and ymax values of an MBR type. The values are returned in the *mbrBuffer* as four double precision values. The caller of the function needs to allocate sufficient memory for the *mbrBuffer* buffer to hold four double precision values.

### Example: Using FNC calls to get an MBR type value

The following function takes an MBR type parameter and returns an array of four double precision values representing the xmin, ymin, xmax, and ymax values that define the MBR.

```

CREATE TYPE MBR_ARRAY AS DOUBLE PRECISION ARRAY[4];
CREATE FUNCTION mbrUDF1(P1 MBR)

```

```

RETURNS MBR_ARRAY
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!mbrUDF1!mbrUDF1.c';

void mbrUDF1 (GEO_HANDLE* mbr_handle,
              ARRAY_HANDLE* resultArray,
              int* indicator_thismbr,
              int* indicator_Result,
              char      sqlstate[6],
              SQL_TEXT  extname[129],
              SQL_TEXT  specific_name[129],
              SQL_TEXT  error_message[257] )
{
    double* mbrBuffer;
    int returnSize;
    int mbrBufferSize;
    array_info_t arrayInfo;
    bounds_t arrayScope[FNC_MAXARRAYDIMENSIONS];
    NullBitVecType      *NullBitVector;

    mbrBufferSize = 4 * sizeof(double);
    mbrBuffer = (double*)FNC_Malloc(mbrBufferSize);
    /* Get the MBR value */
    FNC_MBRGetValue(*mbr_handle, mbrBuffer, &returnSize);

    /* get element type information for the ARRAY */
    FNC_GetArrayTypeInfo(*((ARRAY_HANDLE*)resultArray),
                        &arrayInfo,
                        arrayScope);
    /* Allocate a new NullBitVector */
    NullBitVector = (NullBitVecType*)FNC_malloc(1);

    /* Set array elements within the range. */
    FNC_SetArrayElementsWithMultiValues(
        *((ARRAY_HANDLE*)resultArray),
        arrayScope,
        (void*)mbrBuffer,
        mbrBufferSize, NullBitVector, 1);

    *indicator_Result = 0;
    FNC_free(NullBitVector);
}

```

```
FNC_free(mbrBuffer);
}
```

## FNC\_MBRSetValue

Set the value of an MBR (minimum bounding rectangle) type using a buffer of four double precision values. The values represent the xmin, ymin, xmax, and ymax coordinates that define the MBR.

### Syntax

```
void
FNC_MBRSetValue(GEO_HANDLE    mbrHandle,
                double *      mbrValue)
```

### Syntax Elements

#### *mbrHandle*

A handle to an MBR type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

#### *mbrValue*

A pointer to a buffer containing the four double precision values.

### Usage Notes

FNC\_MBRSetValue sets the value of an MBR type. A buffer *mbrValue* containing four double precision values is provided as input to the routine along with the MBR handle.

### Example: Using FNC calls to set an MBR type value

The following function takes four double precision values as input, and returns an MBR value.

```
CREATE FUNCTION mbrUDF1(P0 DOUBLE PRECISION, P1 DOUBLE PRECISION,
                      P2 DOUBLE PRECISION, P3 DOUBLE PRECISION)
RETURNS MBR
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!mbrUDF1!mbrUDF1.c';

void mbrUDF1 (DOUBLE_PRECISION* p0,
             DOUBLE_PRECISION* p1,
             DOUBLE_PRECISION* p2,
```

```

        DOUBLE_PRECISION* p3,
        GEO_HANDLE* resultMBR,
        int* indicator_p0,
        int* indicator_p1,
        int* indicator_p2,
        int* indicator_p3,
        int* indicator_Result,
        char          sqlstate[6],
        SQL_TEXT      extname[129],
        SQL_TEXT      specific_name[129],
        SQL_TEXT      error_message[257] )
{
    double* mbrBuffer;
    int returnSize;
    int mbrBufferSize;

    mbrBufferSize = 4 * sizeof_DOUBLE_PRECISION;
    mbrBuffer = (double*)FNC_Malloc(mbrBufferSize);
    /* Set the double precision values in the buffer */
    mbrBuffer[0] = *p0;
    mbrBuffer[1] = *p1;
    mbrBuffer[2] = *p2;
    mbrBuffer[3] = *p3;

    /* Set the MBR value */
    FNC_MBRSetValue(*resultMBR, mbrBuffer);

    *indicator_Result = 0;
    FNC_free(mbrBuffer);
}

```

## FNC\_malloc

Allocates a block of memory.

Returns a pointer to a block of at least the specified size in bytes, aligned for use by the UDF, UDM, table operator, contract function, or external stored procedure.

## Syntax

```

void *
FNC_malloc(size_t  size)

```

## Syntax Elements

### *size*

the number of bytes of memory to allocate.

## Usage Notes

The `sqltypes_td.h` header file redefines *malloc* to call *FNC\_malloc*.

WHEN you ...	THEN ...
develop, test, and debug a UDF, UDM, or external stored procedure standalone	include the <code>malloc.h</code> header file and use the standard <i>malloc</i> and <i>free</i> C library functions.
install the source code on the server	do not include <i>malloc.h</i> and use the definitions of <i>malloc</i> and <i>free</i> from the <code>sqltypes_td.h</code> header file. Note that the definitions are used when installing source code on the server, but not when installing objects.

*FNC\_malloc* and *FNC\_free* check to make sure the UDF, UDM, table operator, contract function, or external stored procedure releases all memory before exiting and gives an exception on the transaction if the UDF, UDM, table operator, contract function, or external stored procedure does not release all temporary memory it allocated. This is to prevent memory leaks in the database.

A table function that calls *FNC\_malloc* to allocate memory does not have to call *FNC\_free* to release the memory until at least the `TBL_END` or `TBL_ABORT` phase. This means that the table function must use the *FNC\_TblAllocCtx* and *FNC\_TblGetCtx* calls to save the address that *FNC\_malloc* returns in the general scratch pad.

The maximum amount of memory that any one external routine can allocate is a global configuration setting, where the default is 32MB.

To view or change the memory allocation limit, you can use the *cufconfig* utility. For example, to view the memory allocation limit, use the `-o` option of *cufconfig* and find the setting for the `MallocLimit` field:

```
cufconfig -o
```

For information on *cufconfig* and the `MallocLimit` field, see *Teradata Vantage™ - Database Utilities*, B035-1102.

This does not necessarily guarantee that an external routine can allocate the maximum amount of memory at all times. For performance reasons, an external routine should allocate as little memory as possible. Remember that UDFs and table operators execute in parallel on the database. An all-AMP query that includes a UDF or a table operator allocating 10MB for one instance can use up to 1GB for all instances on a 100 AMP system for just one transaction.



If you circumvent the logic to call *malloc* and *free* directly, then there is a good chance that memory leaks could occur even if the UDF, UDM, table operator, contract function, or external stored procedure frees up memory correctly. The reason is that a UDF, UDM, table operator, contract function, or external stored procedure can abort while it is running if a user aborts the transaction, or if the transaction aborts because of some constraint violation that might occur on another node unbeknownst to the running UDF, UDM, table operator, contract function, or external stored procedure.

## FNC\_SetActivityCount

Sets the number of rows exported.

### Syntax

```
void
FNC_SetActivityCount(int    stream,
                    long    rowsexported)
```

### Syntax Elements

#### *stream*

IN parameter

Specifies which stream to write to.

#### *rowsexported*

IN parameter

The value to be written to ActivityCount.

### Usage Notes

This routine is callable on an AMP vproc only by a table operator.

## FNC\_SetArrayElements

Sets the value of one or more elements of an ARRAY type that is defined to be a return value to a UDF or UDM, or an INOUT or OUT parameter to an external stored procedure. All of the requested elements are set to the same new value.

### Syntax

```
void
FNC_SetArrayElements ( ARRAY_HANDLE    aryHandle,
                      bounds_t         *arrayInterval,
```

```

void      *newValue,
int       nullIndicator,
long      length  )

```

## Syntax Elements

### ***aryHandle***

the handle to an ARRAY type that is defined to be a return value to a UDF or UDM, or an INOUT or OUT parameter to an external stored procedure.

### ***arrayInterval***

an array of `bounds_t` structures that provides the index to the set of ARRAY elements that will be modified to NULL or *newValue*.

For a 1-D ARRAY, the index to the set of elements is provided in the first cell of the *arrayInterval* array. If the ARRAY type is n-D, then subsequent dimension information is placed in cells 2-5 as needed.

The `bounds_t` structure is defined in `sqltypes_td.h` as:

```

typedef struct bounds_t {
    int lowerBound;
    int upperBound;
} bounds_t;

```

The value of *lowerBound* specifies the first value in the given dimension to be modified, and *upperBound* specifies the last value in the given dimension to be modified.

This range of elements must be sequential for a 1-D ARRAY, and must be specified as a slice or rectangle of elements for an n-D ARRAY. The boundaries (lower and upper) for each dimension must be within the limits defined for the ARRAY type.

### ***newValue***

a pointer to a buffer that `FNC_SetArrayElements` uses to set the values of the affected elements.

### ***nullIndicator***

whether to set the requested elements to NULL:

- If *nullIndicator* is -1, then the elements are set to NULL.
- If *nullIndicator* is 0, then the elements are set to the value pointed to by *newValue*.

### ***length***

the total size in bytes of the new value.

## Usage Notes

FNC\_SetArrayElements takes *aryHandle*, *arrayInterval*, *newValue*, *nullIndicator*, and *length* as input arguments and sets the value of the specified elements to NULL or to the value pointed to by *newValue*.

To define the buffer pointed to by *newValue*, use the C data type that maps to the underlying type of the element. For example, if the ARRAY type is defined with an element type of SQL INTEGER, you can define the buffer like this:

```
INTEGER value;
value = 2048;
```

If the underlying type of the element is a character string, and the *newValue* character string is shorter than the size defined for the type, FNC\_SetArrayElements fills to the right with spaces.

Because character data types allow embedded null characters, do not include null termination characters in the value you pass in for the *length* argument.

To guarantee that the value you pass in for the *length* argument matches the length of the data type, use the data type length macros defined in the `sqltypes_td.h` header file. For a list of the data type length macros, see [FNC\\_SetStructuredAttribute](#).

Be sure to release allocated resources once you have processed the data.

For more information, see [Setting the Value of Array Elements](#).

## Examples of Setting Element Values for a 1-D Array

The following examples are based on the following ARRAY definition:

```
/*Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE phonenumbers_ary AS VARRAY(5) OF CHAR(10);
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];
```

### Example: Set the Value of the Element in Position 2 of the Array

In this example, FNC\_SetArrayElements is called to set the element in position 2 to the value of the *newVal* argument.

```
/* This function sets one single element value. */
void setOneElement ( ARRAY_HANDLE    *phone_ary,
                     CHARACTER_LATIN newVal[10],
                     INTEGER         *result,
                     char             sqlstate[6])
{
```

```

int nullIndicator;
bounds_t arrayRange;
nullIndicator = 0;
arrayRange.lowerBound = 2;
arrayRange.upperBound = 2;
/* Set the element in position 2 to the value of newVal. */
FNC_SetArrayElements(*phone_ary, &arrayRange, (void*)newVal,
    nullIndicator, sizeof_CHARACTER_LATIN_WITH_NULL(10));
...
}

```

### Example: Set the Value of the Elements in Positions 1-3 of the Array

In this example, FNC\_SetArrayElements is called to set the elements in positions 1-3 to the value of the *newVal* argument.

```

/* This function sets the values of a set of consecutive elements. */
void setMultiElement( ARRAY_HANDLE    *phone_ary,
                      CHARACTER_LATIN newVal[10],
                      INTEGER          *result,
                      char              sqlstate[6])
{
    int nullIndicator;
    bounds_t arrayRange;
    nullIndicator = 0;
    arrayRange.lowerBound = 1;
    arrayRange.upperBound = 3;

    /* Set the elements in positions 1-3 to the value of newVal. */
    FNC_SetArrayElements(*phone_ary, &arrayRange, (void*)newVal,
        nullIndicator, sizeof_CHARACTER_LATIN_WITH_NULL(10));
    ...
}

```

### Example: Set the Value of All Elements in the Array

In this example, FNC\_SetArrayElements is called to set all of the elements to the value of the *newVal* argument.

```

/* This function sets all element values in the array. */
void setAllElements( ARRAY_HANDLE    *phone_ary,
                    CHARACTER_LATIN newVal[10],

```

```

        INTEGER      *result,
        char          sqlstate[6])
{
    int nullIndicator;
    bounds_t arrayRange;
    nullIndicator = 0;
    arrayRange.lowerBound = 1;
    arrayRange.upperBound = 5;

    /* Set all elements in the array to the value of newVal. */
    FNC_SetArrayElements(*phone_ary, &arrayRange, (void*)newVal,
        nullIndicator, sizeof_CHARACTER_LATIN_WITH_NULL(10));
    ...
}

```

## Examples of Setting Element Values for an n-D Array

The following examples are based on the following n-D ARRAY definition:

```

/*Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE myArray AS VARRAY(1:20)(1:20) OF CHAR(10);
CREATE TYPE myArray AS integer ARRAY[1:20][1:20];

```

### Example: Set the Value of the Element in Position [2:2][2:2] of the Array

In this example, FNC\_SetArrayElements is called to set the element in position [2:2][2:2] to the value of the newValue argument.

```

/* This function receives a new value (newValue) and assigns it to one */
/* particular element in the array. */

void setOneElement( ARRAY_HANDLE *myArray,
                    INTEGER      newValue,
                    INTEGER      *result,
                    char          sqlstate[6])
{
    int nullIndicator;
    bounds_t *arrayRange;
    array_info_t arrayInfo;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];
    nullIndicator = 0;

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*myArray,
        &arrayInfo,
        arrayScope);

    arrayRange =
        (bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

```

```

/* Set values of arrayRange to correspond to the range [2:2][2:2] */
arrayRange[0].lowerBound = 2;
arrayRange[0].upperBound = 2;
arrayRange[1].lowerBound = 2;
arrayRange[1].upperBound = 2;
/* Set the element in position [2:2][2:2] to the value contained */
/* in newValue. */
FNC_SetArrayElements(*myArray, &arrayRange, &newValue,
    nullIndicator, sizeof_INTEGER);
...
}

```

### Example: Set the Value of the Elements in Positions [10:15][10:20] of the Array

In this example, FNC\_SetArrayElements is called to set the elements within positions [10:15][10:20] to the value of the *newValue* argument.

```

/* This function sets the values of a set of elements. */

void setElements( ARRAY_HANDLE *myArray,
                  INTEGER      newValue,
                  INTEGER      *result,
                  char          sqlstate[6])
{
    int nullIndicator;
    bounds_t *arrayRange;
    array_info_t arrayInfo;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];
    nullIndicator = 0;

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*myArray,
                        &arrayInfo,
                        arrayScope);

    arrayRange =
    (bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

    /* Set values of arrayRange to correspond to the */
    /* range [10:15][10:20] */

    arrayRange[0].lowerBound = 10;
    arrayRange[0].upperBound = 15;
    arrayRange[1].lowerBound = 10;

```

```

arrayRange[1].upperBound = 20;

/* Set the element values within the positions [10:15][10:20] to */
/* the value contained in newValue. */
FNC_SetArrayElements(*myArray, &arrayRange, &newValue,
    nullIndicator, sizeof_INTEGER);
...
}

```

## FNC\_SetArrayElementsWithMultiValues

Sets the value of one or more elements of an ARRAY type that is defined to be a return value to a UDF or UDM, or an INOUT or OUT parameter to an external stored procedure. The requested elements can be set to different values as specified by the input.

### Syntax

```

void
FNC_SetArrayElementsWithMultiValues (ARRAY_HANDLE    aryHandle,
                                     bounds_t         *arrayInterval,
                                     void             *newValues,
                                     long             newValuesBufSize,
                                     NullBitVecType   *nullBitVec,
                                     long             nullBitVecBufSize)

```

### Syntax Elements

#### *aryHandle*

the handle to an ARRAY type that is defined to be a return value to a UDF or UDM, or an INOUT or OUT parameter to an external stored procedure.

#### *arrayInterval*

an array of `bounds_t` structures that provides the index to the set of ARRAY elements that will be modified to *newValues*.

For a 1-D ARRAY, the index to the set of elements is provided in the first cell of the *arrayInterval* array. If the ARRAY type is n-D, then subsequent dimension information is placed in cells 2-5 as needed.

The `bounds_t` structure is defined in `sqltypes_td.h` as:

```

typedef struct bounds_t {
    int lowerBound;

```

```
    int upperBound;
} bounds_t;
```

The value of *lowerBound* specifies the first value in the given dimension to be modified, and *upperBound* specifies the last value in the given dimension to be modified.

This range of elements must be sequential for a 1-D ARRAY, and must be specified as a slice or rectangle of elements for an n-D ARRAY. The boundaries (lower and upper) for each dimension must be within the limits defined for the ARRAY type.

### ***newValues***

a pointer to a buffer that FNC\_SetArrayElementsWithMultiValues uses to set the values of the affected elements.

### ***newValuesBufSize***

the total size in bytes of the *newValues* buffer.

### ***nullBitVec***

a pointer to a NullBitVector array previously allocated by the caller. For the range of elements requested, the relevant bits of *nullBitVec* will be set to:

- 1 if the element is present and non-null.
- 0 if the element is present but is set to NULL.

### ***nullBitVecBufSize***

the size of the NullBitVector as allocated by the caller prior to initialization of the NullBitVector by setting all bytes to 0.

This parameter is required for protected mode execution of FNC\_SetArrayElementsWithMultiValues.

## **Usage Notes**

You can call FNC\_GetArrayElements first to get multiple values from an ARRAY and a corresponding NullBitVector. Then you can optionally modify both of these structures and use FNC\_SetArrayElementsWithMultiValues to update the same range of elements in a different ARRAY that has the same ARRAY data type.

Alternatively, you can create your own *newValues* buffer and update an ARRAY using the values from the buffer as follows:

1. Allocate the *newValues* buffer to be the size corresponding to the number of elements to be modified, based on the range specified in the *arrayInterval* input parameter. See [Allocating the newValues Buffer](#).



2. Fill the buffer with the desired element value for each element in the specified range, in row-major order. See [Filling the newValues Buffer](#). You must allocate, initialize and set a corresponding NullBitVector to accompany the values in the *newValues* buffer. See [Setting the NullBitVector](#). Then you can use `FNC_SetArrayElementsWithMultiValues` to update a range of values all at once.

This function provides performance benefits by allowing you to update potentially large sections of an ARRAY with few FNC calls. Otherwise, you would need to call `FNC_SetArrayElements` multiple times in order to set several individual elements to different new values.

For more information, see [Setting the Value of Array Elements](#).

## Allocating the *newValues* Buffer

To define the buffer pointed to by *newValues*, use the C data type that maps to the underlying type of the element. Be sure to allocate the buffer for the number of elements that you are modifying. For example, if the ARRAY type is defined with an element type of SQL INTEGER, you can define the buffer like this:

```
INTEGER value;
value = 2048;
```

If the underlying type of the element is a character string, and the *newValues* character string is shorter than the size defined for the type, `FNC_SetArrayElementsWithMultiValues` fills to the right with spaces.

Because character data types allow embedded null characters, do not include null termination characters in the value you pass in for the *newValuesBufSize* argument.

To guarantee that the value you pass in for the *newValuesBufSize* argument matches the length of the data type, use the data type length macros defined in the `sqltypes_td.h` file when calculating the size of the *newValues* buffer. For a list of the data type length macros, see [FNC\\_SetStructuredAttribute](#).

Be sure to release allocated resources after you process the data.

## Filling the *newValues* Buffer

After you allocate the *newValues* buffer, fill the buffer with the desired element value for each element in the specified range, in row-major order. For each element to be modified, fill the buffer as follows:

1. The first 4 bytes should describe the size of the data. This is only applicable for variable-length element data types.
2. The remaining bytes allocated for each element are allocated to hold the maximum size of the element data type. This space should contain the new element value.

Therefore, for each element, space is allocated as:

- (MAX\_SIZE\_OF\_ELEMENT\_DATA\_TYPE + 4 bytes) for variable-length element data types.
- (MAX\_SIZE\_OF\_ELEMENT\_DATA\_TYPE) for fixed-length data types.

## Setting the NullBitVector

Use `FNC_SetNullBitVector` or `FNC_SetNullBitVectorByElemIndex` to set the bits of *nullBitVec* that correspond to the specified range of elements. Set the relevant bits to the following values:

- 1 to indicate that the corresponding element is present and not NULL.
- 0 to indicate that the corresponding element is present but is set to NULL.

For more information, see [FNC\\_SetNullBitVector](#), [FNC\\_SetNullBitVectorByElemIndex](#), and [Checking and Setting the NullBitVector](#).

## Example Using FNC\_SetArrayElementsWithMultiValues

The following example is based on the following ARRAY definition:

```
/*Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE phonenumbers_ary AS VARRAY(5) OF CHAR(10);
CREATE TYPE phonenumbers_ary AS CHAR(10) ARRAY[5];

/* This function sets the values of a set of consecutive elements from */
/* a source array phone_ary1 to the same range of values in phone_ary2.*/

void setMultiElement ( ARRAY_HANDLE    *phone_ary1,
                      ARRAY_HANDLE    *phone_ary2,
                      INTEGER          *result,
                      char              sqlstate[6])
{
    long length;
    long bufferSize = sizeof_CHARACTER_LATIN_WITH_NULL(10) * 5;
    char *resultBuf;
    NullBitVecType *NullBitVector;
    array_info_t arrayInfo;
    long nullVecBufSize;
    bounds_t *arrayRange;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*phone_ary1,
                        &arrayInfo,
                        arrayScope);
    nullVecBufSize = arrayInfo.totalNumElements;

    resultBuf = (char*)FNC_malloc(bufferSize);
```

```

arrayRange =
(bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

/* Allocate a new NullBitVector to pass to FNC_GetArrayElements. */
NullBitVector = (NullBitVecType*) FNC_malloc (nullVecBufSize);

/* Set all members of NullBitVector to 0. */
memset(NullBitVector, 0, nullVecBufSize);

/* Set values of arrayRange to correspond to the range [1:3] */
arrayRange[0].lowerBound = 1;
arrayRange[0].upperBound = 3;

/*Get elements within the range [1:3] of phone_ary1. */
FNC_GetArrayElements(*phone_ary1, arrayRange, resultBuf, bufferSize,
    NullBitVector, nullVecBufSize, &length);

/* Set the elements in positions 1-3 of phone_ary2 to the same */
/* values as the corresponding elements in phone_ary1. */
FNC_SetArrayElementsWithMultiValues(*phone_ary2, arrayRange,
    (void*)resultBuf, length, NullBitVector, nullVecBufSize);
...
}

```

## Example Using FNC\_SetArrayElementsWithMultiValues

The following example is based on the following n-D ARRAY definition:

```

/*Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE myArray AS VARRAY(1:20)(1:20) OF integer;
CREATE TYPE myArray AS integer ARRAY[1:20][1:20];

/* This function sets the values of a set of elements from a source */
/* ARRAY myArray1 to the same positions of values in myArray2.*/

void setElements ( ARRAY_HANDLE    *myArray1,
                  ARRAY_HANDLE    *myArray2,
                  INTEGER          *result,
                  char             sqlstate[6])
{
    bounds_t *arrayRange;
    array_info_t arrayInfo;
    long length;

```

```

long bufferSize = sizeof_integer * 4;
int *resultBuf;
NullBitVecType *NullBitVector;
long nullVecBufSize;
bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];

/* Call FNC_GetArrayTypeInfo first to find out the number of */
/* elements in myArray1. */
FNC_GetArrayTypeInfo(*myArray1,
                    &arrayInfo,
                    arrayScope);
nullVecBufSize = arrayInfo.totalNumElements;

resultBuf = (int*)FNC_malloc(bufferSize);
arrayRange =
(bounds_t*)FNC_malloc(sizeof(bounds_t)*arrayInfo.numDimensions);

/* Allocate a new NullBitVector to pass to FNC_GetArrayElements. */
NullBitVector = (NullBitVecType*) FNC_malloc (nullVecBufSize);

/* Set all members of NullBitVector to 0. */
memset(NullBitVector, 0, nullVecBufSize);

/* Set values of arrayRange to correspond to the */
/* range [1:2][2:3] */
arrayRange[0].lowerBound = 1;
arrayRange[0].upperBound = 2;
arrayRange[1].lowerBound = 2;
arrayRange[1].upperBound = 3;

/*Get elements within the range [1:2][2:3] of myArray1. */
FNC_GetArrayElements(*myArray1, arrayRange, resultBuf, bufferSize,
                    NullBitVector, nullVecBufSize, &length);

/* Set the element values within the same positions of myArray2 */
/* with the values from myArray1. */
FNC_SetArrayElementsWithMultiValues(*myArray2, &arrayRange,
    (void*)resultBuf, bufferSize, NullBitVector, nullVecBufSize);
...
}

```

## FNC\_SetDatasetLob

This function allows a user to pass a LOB\_LOCATOR to a DATASET data type instance. The LOB\_LOCATOR references a UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value.

### Syntax

```
void
FNC_SetDatasetLob( DATASET_HANDLE  datasetHandle,
                   LOB_LOCATOR     instance  )
```

### Syntax Elements

#### *datasetHandle*

A handle to a DATASET data type instance that is defined to be an input parameter to an external routine.

#### *instance*

A LOB\_LOCATOR which references the schema and data that will be used by a DATASET instance.

### Usage Notes

This function allows a user to pass a LOB\_LOCATOR to a DATASET data type instance. The LOB\_LOCATOR references a UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value. The instance uses this data to set its schema and binary-encoded Avro value. Note that any invalid data provided results in an error.

## Example: FNC\_GetDatasetResultLob and FNC\_SetDatasetLob

### Example Setup

This example references the following table and data.

```
CREATE TABLE datasetTable(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

avro01.data:

```
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002 |1|base16
```

```
.import vartext file avro01.data
USING (avroData VARCHAR(1000), id varchar(10), encoding VARCHAR(20))
INSERT INTO datasetTable (cast(:id AS
INTEGER),cast(TO_BYTES(:avroData, :encoding) AS DATASET STORAGE FORMAT AVRO));
```

### Example Using FNC\_GetDatasetResultLob and FNC\_SetDatasetLob

```
CREATE FUNCTION CreateDATASETlob(a1 TD_ANYTYPE)
RETURNS TD_ANYTYPE
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!CreateDATASETlob!CreateDATASETlob.c!F!CreateDATASETlob';
```

CreateDATASETlob.c:

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>

#define buffer_size 64000

void CreateDATASETlob (DATASET_HANDLE *input,
                      DATASET_HANDLE *result,
                      int *inputNullIndicator,
                      int *outputNullIndicator,
                      char sqlstate[6],
                      SQL_TEXT extname[129],
                      SQL_TEXT specific_name[129],
                      SQL_TEXT error_message[257])
{
    LOB_LOCATOR inLOB;
    LOB_RESULT_LOCATOR outLOB;
    BYTE buffer[buffer_size];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen, actualInputLength;
    int trunc_err = 0;
    FNC_GetDatasetInputLob(*input,&inLOB);
    FNC_GetDatasetResultLob(*result,&outLOB);
```

```

        readlen=0;
        writelen=0;
        actualInputLength = 0;

        FNC_LobOpen(inLOB, &id, 0, 0);
        while( FNC_LobRead(id, buffer, buffer_size, &readlen) == 0 && !
trunc_err )
        {
            trunc_err = FNC_LobAppend(outLOB, buffer, readlen, &writelen);
        }
        FNC_LobClose(id);

        FNC_SetDatasetLob(*result,outLOB);
        sprintf(sqlstate, "00000\0");
        *outputNullIndicator = 0;
    }

```

The following is sample output that shows the text representation of the data:

```

SELECT (CreateDATASETlob(avroFile) RETURNS DATASET STORAGE FORMAT Avro).toJSON()
FROM datasetTable;

>      {"a":1}

```

## FNC\_SetDistinctValue

Set the value of a non-LOB distinct type that is defined to be the return value of a UDF, UDM, or external stored procedure.

For information on setting the value of a distinct type that represents a LOB, see [FNC\\_GetDistinctResultLob](#).

## Syntax

```

void
FNC_SetDistinctValue ( UDT_HANDLE  udtHandle,
                      void         *newValue,
                      int           length )

```

### Syntax Elements

#### *udtHandle*

the handle to a distinct UDT that is defined to be the return value for a UDF or UDM or an INOUT or OUT parameter for an external stored procedure.

***newValue***

a pointer to a buffer containing the new value for the distinct type.

***length***

the size in bytes of the *newValue* buffer.

## Usage Notes

To define the buffer pointed to by *newValue*, use the C data type that maps to the underlying type of the distinct UDT. For example, if the distinct type represents an SQL INTEGER data type, you can define the buffer like this:

```
INTEGER value;
value = 2048;
```

For information on the C data types that you can use, see [C Data Types](#).

If the underlying type of the distinct UDT is a character string, and the *newValue* character string is shorter than the size defined for the type, FNC\_SetDistinctValue fills to the right with spaces.

Because character data types allow embedded null characters, do not include null termination characters in the value you pass in for the *length* argument.

To guarantee that the value you pass in for the *length* argument matches the length of the data type, use these macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_CHARACTER_LATIN( <i>len</i> ) SIZEOF_CHARACTER_KANJISJIS( <i>len</i> ) SIZEOF_CHARACTER_KANJI1( <i>len</i> ) SIZEOF_CHARACTER_UNICODE( <i>len</i> )	Returns the length in bytes of a CHARACTER data type of <i>len</i> characters. For example, the following returns a length of 6 (3 * 2 = 6):  SIZEOF_CHARACTER_UNICODE(3)
SIZEOF_VARCHAR_LATIN( <i>len</i> ) SIZEOF_VARCHAR_KANJISJIS( <i>len</i> ) SIZEOF_VARCHAR_KANJI1( <i>len</i> ) SIZEOF_VARCHAR_UNICODE( <i>len</i> )	Returns the length in bytes of a VARCHAR data type of <i>len</i> characters. For example, the following returns a length of 6 (3 * 2 = 6):  SIZEOF_VARCHAR_UNICODE(3)
SIZEOF_BYTE( <i>len</i> ) SIZEOF_VARBYTE( <i>len</i> )	Returns the length in bytes of the specified BYTE or VARBYTE data type, where <i>len</i> specifies the number of values.
SIZEOF_GRAPHIC( <i>len</i> ) SIZEOF_VARGRAPHIC( <i>len</i> )	Returns the length in bytes of the specified CHARACTER(n) CHARACTER SET GRAPHIC or VARCHAR(n) CHARACTER SET GRAPHIC data type, where <i>len</i> specifies the number of values.



Macro	Description
SIZEOF_BYTEINT SIZEOF_SMALLINT SIZEOF_INTEGER SIZEOF_BIGINT SIZEOF_REAL SIZEOF_DOUBLE_PRECISION SIZEOF_FLOAT SIZEOF_DECIMAL1 SIZEOF_DECIMAL2 SIZEOF_DECIMAL4 SIZEOF_DECIMAL8 SIZEOF_DECIMAL16 SIZEOF_NUMERIC1 SIZEOF_NUMERIC2 SIZEOF_NUMERIC4 SIZEOF_NUMERIC8 SIZEOF_NUMERIC16 SIZEOF_NUMBER	Returns the length in bytes of the specified numeric data type. For NUMBER, the length returned is $4 + 2 + 17 = 23$ bytes since Vantage allocates max length (17 bytes) for the mantissa.
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.
SIZEOF_INTERVAL_YEAR SIZEOF_IntrvlYtoM SIZEOF_INTERVAL_MONTH SIZEOF_INTERVAL_DAY SIZEOF_IntrvlDtoH SIZEOF_IntrvlDtoM SIZEOF_IntrvlDtoS SIZEOF_HOUR SIZEOF_IntrvlHtoM SIZEOF_IntrvlHtoS SIZEOF_MINUTE SIZEOF_IntrvlMtoS SIZEOF_IntrvlSec	Returns the length in bytes of the specified Interval type.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example Using FNC\_SetDistinctValue

```
void meters_t_toFeet( UDT_HANDLE    *metersUdt,
                    UDT_HANDLE    *resultFeetUdt,
                    char          sqlstate[6])
{
    FLOAT value;
    int length;

    /* Get the value of metersUdt. */
    FNC_GetDistinctValue(*metersUdt, &value, sizeof_FLOAT, &length);

    /* Convert meters to feet and set the result value */
    value *= 3.28;
    FNC_SetDistinctValue(*resultFeetUdt, &value, sizeof_FLOAT);    ...
}
```

## FNC\_SetInternalValue

Set the value of a Period, JSON, or DATASET type that is defined to be the result of an external routine.

### Syntax

```
void
FNC_SetInternalValue ( int      typeHandle,
                     void      *newValue,
                     int       length )
```

### Syntax Elements

#### *typeHandle*

The handle to a Period, JSON, or DATASET type instance that is defined to be the return value for a UDF or UDM or an INOUT or OUT parameter for an external stored procedure.

#### *newValue*

A pointer to a buffer containing the new value for the Period, JSON, or DATASET type.

#### *length*

The size in bytes of the *newValue* buffer.

## Usage Notes

### JSON Data Type

You can use `FNC_SetInternalValue` to set a character or binary JSON value. When setting the string representation of a JSON type instance, the string is either UNICODE or LATIN text, depending on how the instance was defined.

Before calling `FNC_SetInternalValue`, the external routine must allocate the buffer pointed to by *newValue*. *newValue* must point to a valid memory buffer that contains the new JSON data to be set. The length of the data cannot exceed the maximum inline length reported by `FNC_GetExtendedJSONInfo`.

Use `FNC_SetInternalValue` only when the JSON data will *not* be stored as a LOB. If `FNC_GetJSONInfo` or `FNC_GetExtendedJSONInfo` returns `numLobs = 0`, you can use `FNC_SetInternalValue`; otherwise, you should use `FNC_GetJSONResultLob` instead.

### DATASET Data Type

Before calling `FNC_SetInternalValue`, the external routine must allocate the buffer pointed to by *newValue*. *newValue* must point to a valid memory buffer that contains the new DATASET data to be set. The length of the data cannot exceed the maximum inline length reported by `FNC_GetDatasetInfo`.

Use `FNC_SetInternalValue` only when the DATASET data will *not* be stored as a LOB. If `FNC_GetDatasetInfo` returns `dataLob = 0`, you can use `FNC_SetInternalValue`; otherwise, you should use `FNC_GetDatasetResultLob` instead.

For Avro instances, this routine allows a user to write the UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value to a buffer and pass it to the database. This is equivalent to the transform format. Failure to write the data in this format results in an error.

A CSV value passed to this function cannot include any optional schema.

### Period Types

For Period types, the format of the return data depends on the element type of the Period data type. If the Period type is:

- `PERIOD(DATE)`, then the format of the *newValue* data is two consecutive DATE values.
- `PERIOD(TIME)`, then the format of the *newValue* data is two consecutive TIME values.
- `PERIOD(TIME WITH TIME ZONE)`, then the format of the *newValue* data is two consecutive TIME WITH TIME ZONE values.
- `PERIOD(TIMESTAMP)`, then the format of the *newValue* data is two consecutive TIMESTAMP values.
- `PERIOD(TIMESTAMP WITH TIME ZONE)`, then the format of the *newValue* data is two consecutive TIMESTAMP WITH TIME ZONE values.

The size of the buffer pointed to by *newValue* must be large enough to hold two consecutive values of the element type of the Period data type. For example, if the Period type is `PERIOD(DATE)`, the buffer must be large enough to hold two consecutive DATE values.

To guarantee that the value you pass in for the *length* parameter matches the length of the data type, use the following macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.

## Restrictions

An external stored procedure that uses CLIV2 to execute SQL must wait for any outstanding CLIV2 requests to complete before calling this function.

## Example of Setting the Value of a Period Type

```
void set_duration( DATE      *date1,
                  DATE      *date2,
                  PDT_HANDLE *result,
                  int        *date1IsNull,
                  int        *date2IsNull,
                  int        *resultIsNull,
                  char        sqlstate[6])
{
    SQL_TEXT extname[129],
             specific_name[129],
             error_message[257] )

{
    DATE new_duration[2];

    /* Set the value of the PERIOD(DATE) result. */
    if (*date2 > *date1)
    {
        new_duration[0] = *date1;
        new_duration[1] = *date2;
    }
    else if (*date1 > *date2)
    {
        new_duration[0] = *date2;
        new_duration[1] = *date1;
    }
    else

```

```

{
    strcpy(sqlstate, "22023");
    strcpy((char *) error_message,
        "PERIOD element values cannot be equal." ) ;
    *resultIsNull = -1;
    return;
}
FNC_SetInternalValue(*result, &new_duration[0], sizeof_DATE*2);

...

}

```

## Example of Setting the Value of a JSON Type

This example uses FNC\_SetInternalValue to create a JSON instance from a string.

SQL definition:

```

REPLACE FUNCTION setJSONValue (a1 VARCHAR(100))
RETURNS JSON(100)
NO SQL
PARAMETER STYLE TD_GENERAL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!setJSONValue!setJSONValue.c!F!setJSONValue';

```

C function definition, setJSONValue.c:

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>

void setJSONValue (VARCHAR_LATIN *jsonstr,
                   JSON_HANDLE *json_instance,
                   char sqlstate[6])
{
    FNC_SetInternalValue((*json_instance),
                        (void*)jsonstr,
                        strlen(jsonstr));
}

```

Example query:

```
SELECT
setJSONValue('{"CompanyName":"Teradata"}').JSONExtractValue('$.CompanyName');
```

Result:

```
setJSONValue('{"CompanyName":"Teradata"}').JSONEXTRACTVALUE(
-----
Teradata
```

## FNC\_SetNullBitVector

Sets either one bit or all bits in a NullBitVector that was previously allocated by the caller.

### Syntax

```
void
FNC_SetNullBitVector ( NullBitVecType *NullBitVector,
                      int             indexValue,
                      int             presenceValue,
                      long             bufSize)
```

### Syntax Elements

#### *NullBitVector*

a NullBitVector array previously allocated by the caller.

The data type used to access the NullBitVector is defined in `sqltypes_td.h` as:

```
typedef unsigned char NullBitVecType;
```

#### *indexValue*

an integer value set to one of the following:

- The single bit to be modified (*indexValue*  $\geq 0$ , as specified in row-major order for an ARRAY).
- -1 to indicate that all bits in the NullBitVector should be modified.

#### *presenceValue*

the value that the presence bit (indicated by *indexValue*) should be set to. If *indexValue* is -1, then all the bits in the NullBitVector will be changed to the value of *presenceValue*. The valid values are 1 or 0.

**bufSize**

the size in bytes of the NullBitVector as allocated by the caller prior to initialization of the NullBitVector by setting all bytes to 0.

**Usage Notes**

FNC\_SetNullBitVector takes *NullBitVector*, *indexValue*, *presenceValue*, and *bufSize* as input and sets the presence bits for one individual bit or for all bits in *NullBitVector*. You can call this function to perform one of the following:

IF you want to...	THEN set <i>indexValue</i> to...	AND set <i>presenceValue</i> to...
set 1 bit to PRESENT	the bit you want to modify	1
set all bits to PRESENT	-1	1
set 1 bit to NOT PRESENT	the bit you want to modify	0
set all bits to NOT PRESENT	-1	0

For more information about using NullBitVectors, see [Checking and Setting the NullBitVector](#).

**Example Using FNC\_SetNullBitVector**

In this example, a new vector is allocated after getting the appropriate information for the ARRAY by calling FNC\_GetArrayTypeInfo. Then, FNC\_SetNullBitVector is called to set all of the bits to "not present".

```
void ArrayUDF ( ARRAY_HANDLE *ary_handle,
                char          sqlstate[6])
{
    NullBitVecType *NullBitVector;
    array_info_t arrayInfo;
    long nullVecBufSize;
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*ary_handle,
                        &arrayInfo,
                        arrayScope);
    (arrayInfo.totalNumElements % 8 == 0) ?
        (nullVecBufSize = arrayInfo.totalNumElements / 8) :
        (nullVecBufSize = arrayInfo.totalNumElements / 8) + 1;

    /* Allocate a new NullBitVector array. */
    NullBitVector = (NullBitVecType*)FNC_malloc(nullVecBufSize);
}
```

```

/* Set all bits in NullBitVector to 0 (not present) */
FNC_SetNullBitVector(NullBitVector, -1, 0, nullVecBufSize);
...
}

```

## FNC\_SetNullBitVectorByElemIndex

Sets one bit in a NullBitVector that was previously allocated by the caller. The bit to be set may be referenced by the ARRAY type element index as specified by dimension.

### Syntax

```

void
FNC_SetNullBitVectorByElemIndex ( NullBitVecType *NullBitVector,
                                int                indexValue[],
                                int                presenceValue,
                                long               bufSize,
                                bounds_t          *arrayScope,
                                int               numDimensions)

```

### Syntax Elements

#### **NullBitVector**

a NullBitVector array previously allocated by the caller.

The data type used to access the NullBitVector is defined in `sqltypes_td.h` as:

```
typedef unsigned char NullBitVecType;
```

#### **indexValue[]**

the index to the ARRAY element whose corresponding presence bit in the NullBitVector is to be set. For a 1-D ARRAY, the index to the last present element is provided as `indexValue[0]`. If the ARRAY type is n-D, then the complete dimension information for this index is placed in cells `indexValue[1]`, `indexValue[2]`, `indexValue[3]` ... `indexValue[FNC_ARRAYMAXDIMENSIONS]` as needed, where `FNC_ARRAYMAXDIMENSIONS` specifies the maximum number of dimensions in an ARRAY type as defined in `sqltypes_td.h`:

```
#define FNC_ARRAYMAXDIMENSIONS 5
```



***presenceValue***

the value that the presence bit (indicated by *indexValue*) should be set to. The valid values are 1 or 0.

***bufSize***

the size in bytes of the NullBitVector as allocated by the caller prior to initialization of the NullBitVector by setting all bytes to 0.

***arrayScope***

an array of `bounds_t` structures that provides the scope information for the ARRAY which the NullBitVector describes. You can call `FNC_GetArrayTypeInfo` to get this information. See [FNC\\_GetArrayTypeInfo \[Deprecated\]](#).

***numDimensions***

the number of dimensions defined for the ARRAY which the NullBitVector describes. You can call `FNC_GetArrayTypeInfo` to get this information. See [FNC\\_GetArrayTypeInfo \[Deprecated\]](#).

## Usage Notes

`FNC_SetNullBitVectorByElemIndex` takes *NullBitVector*, *indexValue*, *presenceValue*, *bufSize*, *arrayScope*, and *numDimensions* as input and sets the presence bits for one individual bit in *NullBitVector*. You can call this function to perform one of the following:

- Set 1 bit to PRESENT by setting *presenceValue* to 1.
- Set 1 bit to NOT PRESENT by setting *presenceValue* to 0.

For more information about using NullBitVectors, see [Checking and Setting the NullBitVector](#).

## Example Using FNC\_SetNullBitVectorByElemIndex

In this example, `memset()` is called to initialize a new NullBitVector by setting all bytes to 0 after it has been allocated by the caller for an ARRAY of type `phonenumbers_ary`. The required information for these operations is retrieved by calling `FNC_GetArrayTypeInfo`. Then, `FNC_SetNullBitVectorByElemIndex` is called to set one of the bits to "not present".

The following example is based on the following 2-D ARRAY definition:

```
/* Oracle-compatible and Teradata syntax respectively: */
CREATE TYPE myArray AS VARRAY(1:20)(1:20) OF INTEGER;
CREATE TYPE myArray AS INTEGER ARRAY[1:20][1:20];

void ArrayUDF ( ARRAY_HANDLE *a,
               char          sqlstate[6])
```

```

{
    NullBitVecType *NullBitVector;
    array_info_t arrayInfo;
    long nullVecBufSize;
    int aryIndex[FNC_ARRAYMAXDIMENSIONS];
    bounds_t arrayScope[FNC_ARRAYMAXDIMENSIONS];

    /* Call FNC_GetArrayTypeInfo first to find out the number of */
    /* elements in the array. */
    FNC_GetArrayTypeInfo(*a,
                        &arrayInfo,
                        arrayScope);
    (arrayInfo.totalNumElements % 8 == 0) ?
        (nullVecBufSize = arrayInfo.totalNumElements / 8) :
        (nullVecBufSize = arrayInfo.totalNumElements / 8) + 1;

    /* Allocate a new NullBitVector array. */
    NullBitVector = (NullBitVecType*)FNC_malloc(nullVecBufSize);

    /* Initialize the NullBitVector to default values */
    memset(NullBitVector, 0, nullVecBufSize);

    /* Set one bit in NullBitVector to 1 (present) */
    aryIndex[0] = 1;
    aryIndex[1] = 1;
    FNC_SetNullBitVectorByElemIndex(NullBitVector, aryIndex, 1,
        nullVecBufSize, arrayScope, arrayInfo.numDimensions);
    ...
}

```

## FNC\_SetStructuredAttribute

Set the non-LOB attribute of a structured type that is defined to be a return value to a UDF, UDM, or external stored procedure.

---

### Note:

For best performance, use [FNC\\_SetStructuredAttributeByNdx](#) instead of this routine. This routine is supported for its ease of use.

---

### Syntax

```

void
FNC_SetStructuredAttribute ( UDT_HANDLE  udtHandle,

```

```

char    *attributePath,
void    *newValue,
int     nullIndicator,
int     length )

```

## Syntax Elements

### ***udtHandle***

the handle to a structured UDT that is defined to be the return value to a UDF or UDM, or an INOUT or OUT parameter to an external stored procedure.

### ***attributePath***

the dot delimited full path to the attribute.

For example, consider a structured UDT called "PersonUDT" that has an attribute called "address" that is an AddressUDT type, which in turn has an attribute called "zipcode". To set the zipcode value, the full path is "address.zipcode".

If the full path to the specified attribute includes a preceding null attribute, FNC\_SetStructuredAttribute returns normally without setting the value of the specified attribute.

### ***newValue***

a pointer to a buffer containing the new value for the attribute.

If the specified attribute is a structured UDT, you can use the *nullIndicator* argument to set the attribute to null, but FNC\_SetStructuredAttribute ignores the *newValue* and *length* arguments.

### ***nullIndicator***

whether to set the attribute to null.

If the value of *nullIndicator* is...

- -1, then FNC\_SetStructuredAttribute sets the attribute to null.  
The *newValue* argument is ignored.
- 0, then FNC\_SetStructuredAttribute sets the attribute to the value pointed to by *newValue*.

### ***length***

the size in bytes of the *newValue* buffer.

If the specified attribute is a structured UDT, you can use the *nullIndicator* argument to set the attribute to null, but FNC\_SetStructuredAttribute ignores the *newValue* and *length* arguments.

## Usage Notes

You can use `FNC_SetStructuredAttribute` to write a string representation of a JSON document to a JSON attribute only if the JSON data is equal to or less than 64000 bytes. If the JSON data is larger than 64000 bytes, you must use `FNC_GetStructuredResultLobAttribute` instead.

To define the buffer pointed to by *newValue*, use the C data type that maps to the underlying type of the attribute. For example, if the distinct type represents an SQL INTEGER data type, you can define the buffer like this:

```
INTEGER value;
value = 2048;
```

For information on the C data types that you can use, see [C Data Types](#).

If the underlying type of the attribute is a character string, and the *newValue* character string is shorter than the size defined for the type, `FNC_SetStructuredAttribute` fills to the right with spaces.

Because character data types allow embedded null characters, do not include null termination characters in the value you pass in for the *length* argument.

To guarantee that the value you pass in for the *length* argument matches the length of the data type in Vantage, use these macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_CHARACTER_LATIN( <i>len</i> ) SIZEOF_CHARACTER_KANJISJIS( <i>len</i> ) SIZEOF_CHARACTER_KANJI1( <i>len</i> ) SIZEOF_CHARACTER_UNICODE( <i>len</i> )	Returns the length in bytes of a CHARACTER data type of <i>len</i> characters. For example, the following returns a length of 6 ( $3 * 2 = 6$ ): SIZEOF_CHARACTER_UNICODE(3)
SIZEOF_VARCHAR_LATIN( <i>len</i> ) SIZEOF_VARCHAR_KANJISJIS( <i>len</i> ) SIZEOF_VARCHAR_KANJI1( <i>len</i> ) SIZEOF_VARCHAR_UNICODE( <i>len</i> )	Returns the length in bytes of a VARCHAR data type of <i>len</i> characters. For example, the following returns a length of 6 ( $3 * 2 = 6$ ): SIZEOF_VARCHAR_UNICODE(3)
SIZEOF_BYTE( <i>len</i> ) SIZEOF_VARBYTE( <i>len</i> )	Returns the length in bytes of the specified BYTE or VARBYTE data type, where <i>len</i> specifies the number of values.
SIZEOF_GRAPHIC( <i>len</i> ) SIZEOF_VARGRAPHIC( <i>len</i> )	Returns the length in bytes of the specified CHARACTER(n) CHARACTER SET GRAPHIC or VARCHAR(n) CHARACTER SET GRAPHIC data type, where <i>len</i> specifies the number of values.
SIZEOF_BYTEINT SIZEOF_SMALLINT SIZEOF_INTEGER SIZEOF_BIGINT SIZEOF_REAL	Returns the length in bytes of the specified numeric data type. For NUMBER, the length returned is $4 + 2 + 17 = 23$ bytes since Vantage allocates max length (17 bytes) for the mantissa.

Macro	Description
SIZEOF_DOUBLE_PRECISION SIZEOF_FLOAT SIZEOF_DECIMAL1 SIZEOF_DECIMAL2 SIZEOF_DECIMAL4 SIZEOF_DECIMAL8 SIZEOF_DECIMAL16 SIZEOF_NUMERIC1 SIZEOF_NUMERIC2 SIZEOF_NUMERIC4 SIZEOF_NUMERIC8 SIZEOF_NUMERIC16 SIZEOF_NUMBER	
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_WZone SIZEOF_TimeStamp SIZEOF_TimeStamp_WZone	Returns the length in bytes of the specified DateTime type.
SIZEOF_INTERVAL_YEAR SIZEOF_IntrvlYtoM SIZEOF_INTERVAL_MONTH SIZEOF_INTERVAL_DAY SIZEOF_IntrvlDtoH SIZEOF_IntrvlDtoM SIZEOF_IntrvlDtoS SIZEOF_HOUR SIZEOF_IntrvlHtoM SIZEOF_IntrvlHtoS SIZEOF_MINUTE SIZEOF_IntrvlMtoS SIZEOF_IntrvlSec	Returns the length in bytes of the specified Interval type.

An external stored procedure that uses CLlv2 to execute SQL must wait for any outstanding CLlv2 requests to complete before calling this function.

### Example Using FNC\_SetStructuredAttribute

```
void setX( UDT_HANDLE *pointUdt,
          INTEGER      *val,
          UDT_HANDLE *resultPoint,
          char         sqlstate[6])
{
    INTEGER x;
    INTEGER newval;
    int nullIndicator;
    int length;
```

```

/* Set the x attribute of the result point. */
nullIndicator = 0;
newval = *val;
FNC_SetStructuredAttribute(*resultPoint, "x", &newval,
                          nullIndicator, sizeof_INTEGER);    ...
}

```

## FNC\_SetStructuredAttributeByNdx

Set the value of a non-LOB attribute of a structured type.

### Syntax

```

void
FNC_SetStructuredAttributeByNdx ( UDT_HANDLE  udtHandle,
                                int           attributeIndex,
                                void          *newValue,
                                int           nullIndicator,
                                int           length )

```

### Syntax Elements

#### *udtHandle*

the handle to a structured UDT.

#### *attributeIndex*

the index of an attribute at the top-most level of the UDT.

The range of values is from 0 to  $i-1$ , where 0 is the index of the first attribute in the UDT and  $i$  is the number of non-nested attributes in the UDT.

For example, consider a structured UDT called PointUDT that has two attributes: the first attribute is called x and the second attribute is called y. To set the x value, use an index of 0. Similarly, to set the y value, use an index of 1.

#### *newValue*

a pointer to a buffer containing the new value for the attribute.

If the specified attribute is a structured UDT, you can use the *nullIndicator* argument to set the attribute to null, but FNC\_SetStructuredAttributeByNdx ignores the *newValue* and *length* arguments.

***nullIndicator***

whether to set the attribute to null.

If the value of *nullIndicator* is...

- -1, then FNC\_SetStructuredAttributeByNdx sets the attribute to null.  
The *newValue* argument is ignored.
- 0, then FNC\_SetStructuredAttributeByNdx sets the attribute to the value pointed to by *newValue*.

***length***

the size in bytes of the *newValue* buffer.

If the specified attribute is a structured UDT, you can use the *nullIndicator* argument to set the attribute to null, but FNC\_SetStructuredAttributeByNdx ignores the *newValue* and *length* arguments.

**Usage Notes**

- To define the buffer pointed to by *newValue*, use the C data type that maps to the underlying type of the attribute. For example, if the distinct type represents an SQL INTEGER data type, you can define the buffer like this:

```
INTEGER value;
value = 2048;
```

For information on the C data types you can use, see [C Data Types](#).

If the underlying type of the attribute is a character string, and the *newValue* character string is shorter than the size defined for the type, FNC\_SetStructuredAttributeByNdx fills to the right with spaces.

Because character data types allow embedded null characters, do not include null termination characters in the value you pass in for the *length* argument.

To guarantee that the value you pass in for the *length* argument matches the length of the data type in Vantage, use these macros defined in the `sqltypes_td.h` header file.

Macro	Description
SIZEOF_CHARACTER_LATIN( <i>len</i> ) SIZEOF_CHARACTER_KANJISJIS( <i>len</i> ) SIZEOF_CHARACTER_KANJI1( <i>len</i> ) SIZEOF_CHARACTER_UNICODE( <i>len</i> )	Returns the length in bytes of a CHARACTER data type of <i>len</i> characters. For example, the following returns a length of 6 ( $3 * 2 = 6$ ): SIZEOF_CHARACTER_UNICODE(3)
SIZEOF_VARCHAR_LATIN( <i>len</i> ) SIZEOF_VARCHAR_KANJISJIS( <i>len</i> )	Returns the length in bytes of a VARCHAR data type of <i>len</i> characters. For example, the following returns a length of 6 ( $3 * 2 = 6$ ):

Macro	Description
SIZEOF_VARCHAR_KANJI1( <i>len</i> ) SIZEOF_VARCHAR_UNICODE( <i>len</i> )	SIZEOF_VARCHAR_UNICODE(3)
SIZEOF_BYTE( <i>len</i> ) SIZEOF_VARBYTE( <i>len</i> )	Returns the length in bytes of the specified BYTE or VARBYTE data type, where <i>len</i> specifies the number of values.
SIZEOF_GRAPHIC( <i>len</i> ) SIZEOF_VARGRAPHIC( <i>len</i> )	Returns the length in bytes of the specified CHARACTER(n) CHARACTER SET GRAPHIC or VARCHAR(n) CHARACTER SET GRAPHIC data type, where <i>len</i> specifies the number of values.
SIZEOF_BYTEINT SIZEOF_SMALLINT SIZEOF_INTEGER SIZEOF_BIGINT SIZEOF_REAL SIZEOF_DOUBLE_PRECISION SIZEOF_FLOAT SIZEOF_DECIMAL1 SIZEOF_DECIMAL2 SIZEOF_DECIMAL4 SIZEOF_DECIMAL8 SIZEOF_DECIMAL16 SIZEOF_NUMERIC1 SIZEOF_NUMERIC2 SIZEOF_NUMERIC4 SIZEOF_NUMERIC8 SIZEOF_NUMERIC16 SIZEOF_NUMBER	Returns the length in bytes of the specified numeric data type. For NUMBER, the length returned is $4 + 2 + 17 = 23$ bytes since Vantage allocates max length (17 bytes) for the mantissa.
SIZEOF_DATE SIZEOF_ANSI_Time SIZEOF_ANSI_Time_wZone SIZEOF_TimeStamp SIZEOF_TimeStamp_wZone	Returns the length in bytes of the specified DateTime type.
SIZEOF_INTERVAL_YEAR SIZEOF_IntrvlYtoM SIZEOF_INTERVAL_MONTH SIZEOF_INTERVAL_DAY SIZEOF_IntrvlDtoH SIZEOF_IntrvlDtoM SIZEOF_IntrvlDtoS SIZEOF_HOUR SIZEOF_IntrvlHtoM SIZEOF_IntrvlHtoS SIZEOF_MINUTE SIZEOF_IntrvlMtoS SIZEOF_IntrvlSec	Returns the length in bytes of the specified Interval type.



- Setting the Value of a Nested Attribute

To set the value of an attribute that is nested in the hierarchy of a UDT that is defined to be a return value to a UDF, UDM, or external stored procedure, follow these steps:

1. Call `FNC_GetStructuredAttributeByNdx` to obtain the UDT handle of the next structured UDT attribute in the path to the target attribute.

Repeat this step, passing in the newly obtained UDT handle as the *udtHandle* argument, *n-2* times, where *n* is the nesting level of the target attribute.

2. Call `FNC_SetStructuredAttributeByNdx`, passing in the UDT handle of the structured UDT attribute that contains the target attribute.

- An external stored procedure that uses CLv2 to execute SQL must wait for any outstanding CLv2 requests to complete before calling this function.

### Example Using `FNC_SetStructuredAttributeByNdx`

```
void setY( UDT_HANDLE *pointUdt,
          INTEGER      *val,
          UDT_HANDLE *resultPoint,
          char          sqlstate[6])
{
    INTEGER newval;
    int nullIndicator;
    int length;

    /* Set the y attribute (second attribute) of the result point. */
    nullIndicator = 0;
    newval = *val;
    FNC_SetStructuredAttributeByNdx(*resultPoint, 1,
    &newval,                                     nullIndicator, sizeof_INTEGER);    ...
}
```

## FNC\_SetVarCharLength

Sets the length, in bytes, for an external stored procedure output parameter or the result of a UDF or UDM that has a VARCHAR data type.

### Syntax

```
void
FNC_SetVarCharLength(void *outputString,
                    int    outputStringLength);
```

## Syntax Elements

### *outputString*

a pointer to an external stored procedure output parameter or the result of a UDF or UDM that has a data type of VARCHAR.

### *outputStringLength*

the length of the string pointed to by *outputString*.

## Usage Notes

Normally, the database uses *strlen* to determine the length of an external routine output parameter or result. If the output string contains NULLs, you can use `FNC_SetVarCharLength` to explicitly set the length of the output string. If you call `FNC_SetVarCharLength`, the database does not use *strlen* to get the length of an external routine output parameter or result.

Undefined results occur if the *outputString* argument points to data that is not VARCHAR or if *outputStringLength* is an incorrect length.

## Example Using `FNC_SetVarCharLength`

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"

void dstrev13(VARCHAR_LATIN *input,
             int          inplen,
             VARCHAR_LATIN *result)
{
    int i;
    for (i=0; i< inplen; i++)
        result[inplen-i-1] = input[i];
}

void strrevdecomp_varchar(VARBYTE      *InputValue,
                          VARCHAR_LATIN *ResultValue,
                          char          sqlstate[6])
{
    dstrev13(InputValue->bytes, InputValue->length, ResultValue);
    FNC_SetVarCharLength(ResultValue, InputValue->length);
}
```

## FNC\_SetXML

Set the value of an XML type.

## Syntax

```
void
FNC_SetXML(XML_HANDLE      xmlHandle,
           byte             *xmlBuffer,
           int              xmlSize)
```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

### *xmlBuffer*

A pointer to the XML value to be set.

### *xmlSize*

The total size in bytes of the XML value.

## Usage Notes

FNC\_SetXML is used to set an XML return value or OUT parameter value using a memory buffer.

The XML handle *xmlHandle* is passed as input along with a pointer to the XML value. The XML value must be in the UNICODE character set.

Note that the XML value that is passed to this routine must contain well-formed XML data, otherwise you will get an error.

FNC\_SetXML can only be called for inline XML values, that is XML values that are less than 64K, otherwise you will get an error.

## Example: FNC\_SetXML

The following UDF takes an XML parameter as input, retrieves the XML value and sets the return XML value with it. It assumes the XML size is small enough to be read using a single read.

```
CREATE FUNCTION xmlUDF3(P1 XML)
RETURNS XML
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF3!xmlUDF3.c';
```

```

void xmlUDF3(XML_HANDLE* xml_handle,
             XML_HANDLE* return_handle,
             int* indicator_thisXML,
             int* indicator_returnValue,
             char    sqlstate[6],
             SQL_TEXT      extname[129],
             SQL_TEXT      specific_name[129],
             SQL_TEXT      error_message[257] )
{
    FNC_XMLSize_t xmlSize;
    byte* xmlBuffer;
    FNC_XMLSize_t xmlBufferSize;
    int numLobs;

    /* Get the XML Size */
    FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);

    if(numLobs ==1 )
    { /* Return null if a LOB is passed in */
        *indicator_returnValue = -1;
        return;
    }

    /* Read the XML value. "+ 2" for the Unicode null termination character. */
    xmlBuffer = (byte*)FNC_Malloc(xmlSize + 2);
    xmlBufferSize = xmlSize + 2;
    FNC_GetXML(*xml_handle,xmlBuffer, xmlBufferSize, &xmlSize);

    /* Set the return value */
    FNC_SetXML(*return_handle, xmlBuffer, xmlSize);
    *indicator_returnValue = 0;
    FNC_free(xmlBuffer);
}

```

## FNC\_SetXMLBlob

Set the value of an XML type using a BLOB locator.

## Syntax

```
void
FNC_SetXMLBlob(XML_HANDLE      xmlHandle,
               LOB_RESULT_LOCATOR xmlBlob)
```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

### *xmlBlob*

A LOB locator for the BLOB containing the return value for the XML type.

## Usage Notes

FNC\_SetXMLBlob is used to set the XML return value or OUT parameter value using a LOB locator.

The XML handle *xmlHandle* is passed as input along with a LOB locator *xmlBlob*. The *xmlBlob* locator value is obtained by calling FNC\_GetXMLResultBlob and LOB FNC routines are used to set the BLOB value. It is then passed to the FNC\_SetXMLBlob routine. UTF-8 encoding should be used to write to the BLOB.

Note that the BLOB passed to this routine must contain well-formed XML data, otherwise you will get an error.

FNC\_SetXMLBlob can only be called for LOB-based XML values, otherwise you will get an error.

## Example: FNC\_GetXMLResultBlob and FNC\_SetXMLBlob

The following function takes an input BLOB value and returns an XML value back to the user.

```
CREATE FUNCTION xmlUDF7(P1 BLOB AS LOCATOR)
RETURNS XML
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF7!xmlUDF7.c';

void xmlUDF7(LOB_LOCATOR* input_blob,
            XML_HANDLE *xml_handle,
            int* indicator_inputBLOB,
            int* indicator_returnXML,
            char sqlstate[6],
```

```

        SQL_TEXT    extname[129],
        SQL_TEXT    specific_name[129],
        SQL_TEXT    error_message[257] )
{
    LOB_RESULT_LOCATOR xmlBlob;
    BYTE buffer[100000];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    int numLobs;
    FNC_XMLSize_t xmlSize;

    /* Get the XML Size */
    FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);

    if(numLobs ==1 )
    {

        /* Read XML return lob locator */
        FNC_GetXMLResultBlob(*xml_handle,&xmlBlob);
        /* Use LOB FNC calls to read the input BLOB into the XML clob
*/

        FNC_LobOpen(*input_blob, &id, 0, 0);
        while( FNC_LobRead(id, buffer, 100000, &actlen) == 0 && !trunc_err)
            trunc_err = FNC_LobAppend(*xmlClob, buffer, actlen, &actlen);
        FNC_LobClose(id);

        /* Set the return XML value */
        FNC_SetXMLBlob(*xml_handle,xmlClob);
        *indicator_returnXML = 0;
    }
}

```

## FNC\_SetXMLByte

Set the value of an XML type. The value passed should be in the UTF-8 encoding.

### Syntax

```

void
FNC_SetXMLByte(XML_HANDLE    xmlHandle,

```

```

byte      *xmlBuffer,
int       xmlSize)

```

## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

### *xmlBuffer*

A pointer to the XML value to be set.

### *xmlSize*

The total size in bytes of the XML value.

## Usage Notes

FNC\_SetXMLByte is used to set the XML return value or OUT parameter value using a memory buffer.

The XML handle *xmlHandle* is passed as input along with a pointer to the XML value. The XML value should be in the UTF-8 encoding.

Note that the XML value that is passed to this routine must contain well-formed XML data, otherwise you will get an error.

FNC\_SetXMLByte can only be called for inline XML values, that is XML values less than 64K, otherwise you will get an error.

## Example: FNC\_GetXMLByte and FNC\_SetXMLByte

The following UDF takes an XML parameter as input, retrieves the XML value in binary form, and sets the return XML value in binary form. It assumes the XML size is small enough to be read using a single read.

```

CREATE FUNCTION xmlUDF6(P1 XML)
RETURNS XML
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF6!xmlUDF6.c';

void xmlUDF6(XML_HANDLE* xml_handle,
             XML_HANDLE* return_handle,
             int* indicator_thisXML,
             int* indicator_returnValue,

```

```

        char    sqlstate[6],
        SQL_TEXT    extname[129],
        SQL_TEXT    specific_name[129],
        SQL_TEXT    error_message[257] )
{
    FNC_XMLSize_t xmlSize;
    byte* xmlBuffer;
    FNC_XMLSize_t xmlBufferSize;
    int numLobs;

    /* Get the XML Size */
    FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);

    if(numLobs ==1 )
    { /* Return null if a LOB is passed in */
        *indicator_returnValue = -1;
        return;
    }

    /* Read the XML value */
    xmlBuffer = (byte*)FNC_Malloc(xmlSize);
    xmlBufferSize = xmlSize;
    FNC_GetXMLByte(*xml_handle,xmlBuffer, xmlBufferSize, &xmlSize);

    /* Set the return value */
    FNC_SetXMLByte(*return_handle, xmlBuffer, xmlSize);
    *indicator_returnValue = 0;
    FNC_free(xmlBuffer);
}

```

## FNC\_SetXMLClob

Set the value of an XML type using a LOB locator.

### Syntax

```

void
FNC_SetXMLClob(XML_HANDLE      xmlHandle,
               LOB_RESULT_LOCATOR xmlClob)

```



## Syntax Elements

### *xmlHandle*

A handle to an XML type that is defined to be a return value for a UDF/UDM or an INOUT/OUT parameter to an external stored procedure.

### *xmlClob*

A LOB locator for the CLOB containing the return value for the XML type.

## Usage Notes

FNC\_SetXMLClob is used to set an XML return value or OUT parameter value using a LOB locator.

The XML handle *xmlHandle* is passed as input along with a LOB locator *xmlClob*. The *xmlClob* locator value is obtained by calling FNC\_GetXMLResultClob and LOB FNC routines are used to set the CLOB value. It is then passed to the FNC\_SetXMLClob routine. The CLOB must be in the UNICODE character set.

Note that the CLOB returned by this routine must contain well-formed XML data, otherwise you will get an error.

FNC\_SetXMLClob can only be called for LOB-based XML values, otherwise you will get an error.

## Example: FNC\_GetXMLResultClob and FNC\_SetXMLClob

The following function takes an input CLOB value and returns an XML value back to the user.

```
CREATE FUNCTION xmlUDF4(P1 CLOB AS LOCATOR)
RETURNS XML
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
LANGUAGE C
EXTERNAL NAME 'CS!xmlUDF4!xmlUDF4.c';

void xmlUDF4(LOB_LOCATOR* input_clob,
             XML_HANDLE *xml_handle,
             int*         indicator_inputCLOB,
             int*         indicator_returnXML,
             char         sqlstate[6],
             SQL_TEXT     extname[129],
             SQL_TEXT     specific_name[129],
             SQL_TEXT     error_message[257] )
{
    LOB_RESULT_LOCATOR xmlClob;
```

```

BYTE buffer[100000];
LOB_CONTEXT_ID id;
FNC_LobLength_t actlen;
int trunc_err = 0;
int numLobs;
FNC_XMLSize_t xmlSize;

/* Get the XML Size */
FNC_GetXMLInfo(*xml_handle, &xmlSize, &numLobs);

if(numLobs ==1 )
{
  /* Read XML return lob locator */
  FNC_GetXMLResultClob(*xml_handle,&xmlClob);

  /* Use LOB FNC calls to read the input CLOB into the XML clob
  */

  FNC_LobOpen(*input_clob, &id, 0, 0);
  while( FNC_LobRead(id, buffer, 100000, &actlen) == 0 && !trunc_err)
    trunc_err = FNC_LobAppend(*xmlClob, buffer, actlen, &actlen);
  FNC_LobClose(id);

  /* Set the return XML value */
  FNC_SetXMLClob(*xml_handle,xmlClob);
  *indicator_returnXML = 0;
}
}

```

## FNC\_TblAbort

Allows a table function or a table operator to gracefully abort a request when it encounters an error condition and cannot continue.

IF the abort is initiated by ...	THEN FNC_TblAbort returns ...
this copy of the table function	1.
another copy of the table function	0.

This library function is valid during any phase or mode in which the table function was invoked.

## Syntax

```
int
FNC_TblAbort(void)
```

## Usage Notes

Because any copy of the table function can call FNC\_TblAbort, the table function must check the return value to determine which copy made the call.

IF the return value is ...	THEN ...
0	<p>another copy of the table function initiated the abort.</p> <p>Upon return of the call to FNC_TblAbort, this copy of the table function should simply return. This copy of the function will be called again and when it calls FNC_GetPhase, the FNC_Phase argument will be set to TBL_ABORT. The function can close all resources and release any allocated memory at that time.</p>
1	<p>this copy of the table function initiated the abort.</p> <p>The other copies of the table function will be notified about the abort the next time they call FNC_GetPhase, at which point the the value of the FNC_Phase return argument will be TBL_ABORT. Control returns to this copy of the table function after all other copies return from the TBL_ABORT phase, at which point the table function should take the following steps:</p> <ol style="list-style-type: none"> <li>1. Set the <i>sqlstate</i> argument to an appropriate SQLSTATE exception condition. For more information, see <a href="#">Returning SQLSTATE Values</a>.</li> <li>2. Set the <i>error_message</i> string to the error message text. The characters must be inside the LATIN character range.</li> <li>3. Close all resources and release any allocated memory.</li> <li>4. Write to an external log or send external error notification messages, if applicable.</li> <li>5. Return.</li> </ol>

This function can only be called from within a table function or a table operator. Calling this function from a UDM, external stored procedure, scalar function, or aggregate function results in an exception on the transaction.

## Example Using FNC\_TblAbort

```
FNC_Phase    Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)    {
    switch(Phase)
    {
        ...
        case TBL_BUILD:
        {
```

```

...
/* Get some bad data here. Need to abort. */
if( FNC_TblAbort() )
{
    /* At this point, all other copies of the function */
    /* should have finished cleaning up */
    strcpy(sqlstate, "U0004");
    strcpy(error_message, "Bad Input data - not processed");
    break;
}
...
}
}
...

```

## FNC\_TblAllocCtrlCtx

Allocates a control scratchpad to propagate data from the table function control copy to all other copies running on all other AMP vprocs.

FNC\_TblAllocCtrlCtx returns the address of the control scratchpad that the control copy of the table function can use to propagate data to all other copies of the table function.

The return value is a NULL pointer if the control copy of the table function already allocated a control scratchpad or the scratchpad could not be allocated.

Use this library function when the return value of FNC\_GetPhase is TBL\_MODE\_CONST, indicating that the SELECT statement invoked the table function with constant expression input arguments. For example:

```

SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;

```

### Syntax

```

void *
FNC_TblAllocCtrlCtx(int length)

```

### Syntax Elements

#### *length*

the size, in bytes, to allocate to the control scratchpad.

The maximum length is 64 KB.

## Usage Notes

Use the control scratchpad to keep track of what a table function is supposed to be doing and what it has left to do.

Calling this function is valid when:

- The table function calls `FNC_GetPhase` and gets a value of `TBL_PRE_INIT` for the processing phase
- This copy of the table function establishes itself as the control copy of the table function by calling `FNC_TblControl`

Do not store pointers in the scratchpad that reference other structures in the scratchpad, because the returned address of the scratchpad is not the same for subsequent invocations of the table function. Instead, use relative addressing, such as offsets from the current address of the scratchpad.

After the control copy of the table function completes the `TBL_PRE_INIT` processing phase, all copies of the table function (including the control copy) can gain access to the data that the control copy of the table function stores in the control scratchpad by calling `FNC_TblGetCtrlCtx` in any processing phase except for the `TBL_PRE_INIT` phase. Subsequent changes to the control scratchpad are considered local and are retained in the scratchpad for the next iterations of the local table function copy.

This function can only be called from within a table function. Calling this function from a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

Calling this function is valid only when the table function calls `FNC_GetPhase` and gets the following return values:

- `TBL_MODE_CONST` for the mode
- `TBL_PRE_INIT` for the processing phase

## Example Using `FNC_TblAllocCtrlCtx`

```
typedef struct {
    unsigned short cntrl_fnc_AMP
    int            qfd;
    ...
} ctrl_ctx;

ctrl_ctx      *options;
AMP_Info_t    *LocalConfig;
FNC_Phase     Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            LocalConfig = FNC_AMPInfo();
```

```

    if ( FNC_TblControl() )
    {
        options = FNC_TblAllocCtrlCtx(sizeof(ctrl_ctx));
        options->ctrl_fnc_AMP = LocalConfig->AMPId;
        ...
    }
}
...
}
...

```

## FNC\_TblAllocCtx

Allocates a general scratchpad to retain data between iterations of a local table function copy.

FNC\_TblAllocCtx returns the address of the general scratchpad that a local table function copy can use to retain data between iterations.

The return value is a NULL pointer if the table function already allocated a general scratchpad or the scratchpad could not be allocated.

This library function is valid in either of the following two modes (returned by the FNC\_GetPhase library function):

- TBL\_MODE\_CONST, when the SELECT statement invoked the table function with constant expression input arguments
- TBL\_MODE\_VARY, when the SELECT statement invoked the table function using the columns from a derived table as input arguments

### Syntax

```

void *
FNC_TblAllocCtx(int length)

```

### Syntax Elements

#### *length*

the size, in bytes, to allocate to the general scratchpad.

The maximum length is 64 KB.

### Usage Notes

Subsequent invocations of the table function can gain access to the scratchpad by calling FNC\_TblGetCtx.

Do not store pointers in the scratchpad that reference other structures in the scratchpad, because the returned address of the scratchpad is not the same for subsequent invocations of the table function. Instead, use relative addressing, such as offsets from the current address of the scratchpad.

The general scratchpad is local to each copy of the table function for the current transaction or request running on each AMP vproc. It is not retained for subsequent transactions or requests.

The control copy of a table function can place a flag in the general scratchpad to remember that it is the control copy.

To get around the 64 KB maximum length of the scratch pad, call `FNC_malloc` and save the address in the scratchpad so that you can refer to it in subsequent calls. Remember to call `FNC_free` during the `FNC_END` or `FNC_ABORT` phase or you will get a memory not freed error.

This function can only be called from within a table function. Calling this function from a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

This function can only be called once.

### Example Using `FNC_TblAllocCtx`

```
typedef struct {
    unsigned short control_flag;
    int           qfd;
    ...
} scratchpad;

scratchpad      *options;
FNC_Phase      Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            if ( FNC_TblControl() )
            {
                options = FNC_TblAllocCtx(sizeof(scratchpad));
                options->control_flag = 1;
                ...
            }
        }
        ...
    }
}
...
```

## FNC\_TblControl

Designates a table function as the controlling copy of all other copies of the table function running on other AMP vprocs.

IF the call is ...	THEN FNC_TblControl returns ...
successful and the copy of table function can take the control role of the table function	1.
not successful and the copy of the table function should not assume the role of the control copy	0.

Use this library function when the return value of FNC\_GetPhase is TBL\_MODE\_CONST, indicating that the SELECT statement invoked the table function with constant expression input arguments.

For example:

```
SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;
```

### Syntax

```
int
FNC_TblControl(void)
```

### Usage Notes

Setting up a controlling copy of a table function is useful when there is a need to distribute certain external control data among various copies of the table function that run on different AMP vprocs, but the external control data is on a particular node that only one copy of the table function can access.

Only one copy of the table function can successfully call FNC\_TblControl.

After calling FNC\_TblControl, the controlling copy of a table function can distribute control data to other table function copies by calling FNC\_TblAllocCtrlCtx to allocate a control scratchpad.

This function can only be called from within a table function. Calling this function from a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

This function can only be called once.

A table function that calls FNC\_TblControl cannot also call FNC\_TblFirstParticipant.

Calling this function is valid only when the table function calls FNC\_GetPhase and gets the following return values:

- TBL\_MODE\_CONST for the FNC\_Mode result



- TBL\_PRE\_INIT for the FNC\_Phase output argument

### Example Using FNC\_TblControl

```
typedef struct {
    unsigned short cntrl_fnc_AMP
    int            qfd;
    ...
} ctrl_ctx;

ctrl_ctx      *options;
AMP_Info_t    *LocalConfig;
FNC_Phase     Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            LocalConfig = FNC_AMPInfo();
            if ( FNC_TblControl() )
            {
                options = FNC_TblAllocCtrlCtx(sizeof(ctrl_ctx));
                options->cntrl_fnc_AMP = LocalConfig->AMPId;
                ...
            }
        }
        ...
    }
}
...
```

## FNC\_TblFirstParticipant

Provides a way to implement a table function that only needs one copy to participate and does not care which AMP the copy runs on.

IF the copy of the table function ...	THEN FNC_TblFirstParticipant returns ...
is the first copy to call FNC_TblFirstParticipant	1. This copy of the function will participate in the current transaction and request.
is not the first copy to call FNC_TblFirstParticipant	0.

IF the copy of the table function ...	THEN FNC_TblFirstParticipant returns ...
calls FNC_TblFirstParticipant in the wrong mode or phase	-1.

Use this library function when the return value of FNC\_GetPhase is TBL\_MODE\_CONST, indicating that the SELECT statement invoked the table function with constant expression input arguments. For example:

```
SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;
```

## Syntax

```
int
FNC_TblFirstParticipant(void)
```

## Usage Notes

Use this library function when it does not matter which copy of the table function participates in the current transaction and request.

Because all copies of the table function call FNC\_TblFirstParticipant, each copy must check the return value to determine how to proceed.

IF the return value is ...	THEN ...
1	this copy of the table function is the one participating in the current transaction and request.
0	another copy of the table function is the one participating in the current transaction and request. When the call to FNC_TblFirstParticipant returns, this copy of the table function should take the following steps: Call FNC_TblOptOut. Return.

This function can only be called from within a table function. Calling this function in a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

This function can only be called once.

A table function that calls FNC\_TblFirstParticipant cannot also call FNC\_TblControl.

Calling this function is valid when the table function calls FNC\_GetPhase and gets a:

- value of TBL\_PRE\_INIT for the processing phase

- return value of TBL\_MODE\_CONST

If one copy of the table function calls FNC\_TblFirstParticipant, then all copies of the table function must also call FNC\_TblFirstParticipant.

## Example Using FNC\_TblFirstParticipant

```
FNC_Phase    Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {

        case TBL_PRE_INIT:

            switch (FNC_TblFirstParticipant() )
            {
                case 1: /* participant */
                    return;
                case 0: /* not participant */
                    if (FNC_TblOptOut())
                        strcpy(sqlstate, "U0006"); /* an error return */
                    return;
                default: /* -1 or other error */
                    strcpy(sqlstate, "U0007");
                    return;
            }
            ...
        }
        ...
    }
}
```

## FNC\_TblGetColDef

Returns the definitions of the result columns that must be returned by a table function with dynamic result row specification.

### Result Type

FNC\_TblGetColDef returns a pointer to an FNC\_ColumnDef\_t structure that contains all the result column definitions.

If the table function that calls FNC\_TblGetColDef is not a table function with dynamic result row specification, FNC\_TblGetColDef returns a NULL pointer.

## Syntax

```
FNC_ColumnDef_t *
FNC_TblGetColDef()
```

### FNC\_ColumnDef\_t

```
typedef struct
{
    int    num_columns;
    parm_t column_types[1];
} FNC_ColumnDef_t;
```

### parm\_t

Defined in sqltypes\_td.h as:

```
typedef struct parm_t
{
    dtype_et    datatype;
    dmode_et    direction;
    charset_et   charset;
    union {
        long    length;
        int     intervalrange;
        int     precision;
        struct {
            int  totaldigit;
            int  fracdigit;
        } range;
    } size;
} parm_t;
```

## Syntax Elements

### *num\_columns*

Number of entries in the *column\_types* array, which is the same as the number of result columns for the table function execution.

### *column\_types*

A *parm\_t* array that provides the data type and attributes of each result column.

***datatype***

Specifies the data type of the column. The `sqltypes_td.h` header file defines `dtype_et` as:

```
typedef int dtype_et;
```

Valid values are defined by the `dtype_en` enumeration in `sqltypes_td.h`. For a list of the valid values, see the *dtype* argument in [FNC\\_CallSP](#).

***direction***

Does not provide information about result column definitions. For details on the `direction` member of the `parm_t` structure, see [FNC\\_CallSP](#).

***charset***

Specifies the character set of CHAR or VARCHAR data.

The `sqltypes_td.h` header file defines `charset_et` as:

```
typedef int charset_et;
```

Valid values are defined by the `charset_en` enumeration in `sqltypes_td.h`:

```
typedef enum charset_en
{
    UNDEF_CT=0,
    LATIN_CT=1,
    UNICODE_CT=2,
    KANJISJIS_CT=3,
    KANJI1_CT=4
} charset_en;
```

***length***

Specifies the length of a CHAR, VARCHAR, or BYTE type.

***intervalrange***

Specifies the range of an INTERVAL type. For example, 4 for INTERVAL YEAR(4).

***precision***

Specifies the precision in a TIME or TIMESTAMP type. For example, 4 for TIME(4).

***totaldigit***

Specifies the value *m* in a DECIMAL(*m*,*n*) or INTERVAL SECOND (*m*,*n*) type.

***fracdigit***

Specifies the value *n* in a DECIMAL(*m*,*n*) or INTERVAL SECOND (*m*,*n*) type.

## Usage Notes

When a table function with dynamic result row specification is defined by the CREATE FUNCTION statement, the RETURNS TABLE VARYING COLUMNS specifies the maximum number of result columns that the table function supports.

The actual number and definition (data type and attributes) of the output columns that the table function must return is not known until runtime, when the table function is invoked in a SELECT statement.

A table function uses FNC\_TblGetColDef at runtime to get the number and definition of the output columns that the table function must return.

This function can be called during any table function processing phase.

This function can only be called from within a table function. Calling this function from an external stored procedure, scalar UDM, scalar function, or aggregate function results in an exception on the transaction.

## Example Using FNC\_TblGetColDef

```
FNC_ColumnDef_t *The_Columns;

The_Columns = FNC_TblGetColDef();
```

## Related Information

FOR more information on ...	SEE ...
defining a table function with dynamic result row specification	the RETURNS TABLE VARYING COLUMNS clause of the CREATE FUNCTION (table form) statement in <i>Teradata Vantage™ - SQL Data Definition Language Detailed Topics</i> , B035-1184.
invoking a table function with dynamic result row specification	the RETURNS clause of the SELECT statement in <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146.

## FNC\_TblGetCtrlCtx

Get the address of a previously-allocated control scratchpad to retain data between iterations of a local table function copy.

FNC\_TblGetCtrlCtx returns the address of the control scratchpad that a local table function copy can use to retain data between iterations.

The return value is a NULL pointer if the general scratchpad has not been previously allocated with a call to `FNC_TblAllocCtrlCtx`.

Use this library function when the return value of `FNC_GetPhase` is `TBL_MODE_CONST`, indicating that the `SELECT` statement invoked the table function with constant expression input arguments.

For example:

```
SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;
```

## Syntax

```
void *
FNC_TblGetCtrlCtx(void)
```

## Usage Notes

To gain access to the data that the control copy of the table function stores in the control scratchpad, use this function when a call to `FNC_GetPhase` returns a value other than `TBL_PRE_INIT` for the processing phase.

Subsequent changes to the control scratchpad are considered local and are retained in the scratchpad for the next iterations of the local table function copy.

Use the control scratchpad to keep track of what a table function is supposed to be doing and what it has left to do.

Do not store pointers in the scratchpad that reference other structures in the scratchpad, because the returned address of the scratchpad is not the same for subsequent invocations of the table function. Instead, use relative addressing, such as offsets from the current address of the scratchpad.

This function can only be called from within a table function.

Calling this function is valid only when the table function calls `FNC_GetPhase` and gets the following return values:

- `TBL_MODE_CONST` for the mode
- a value other than `TBL_PRE_INIT` for the processing phase

## Example Using `FNC_TblGetCtrlCtx`

```
typedef struct {
    unsigned short cntrl_fnc_AMP
    int            qfd;
    ...
}
```

```

} ctrl_ctx;

ctrl_ctx      *options;
FNC_Phase     Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            ...
        }
    }
    case TBL_INIT:
    {
        options = FNC_TblGetCtrlCtx();
        ...
    }
    ...
}
}
...

```

## FNC\_TblGetCtx

Get the address of a previously-allocated general scratchpad to retain data between iterations of a local table function copy.

FNC\_TblGetCtx returns the address of the general scratchpad that a local table function copy can use to retain data between iterations.

The return value is a NULL pointer if the general scratchpad has not been previously allocated with a call to FNC\_TblAllocCtx.

This library function is valid during any mode or processing phase in which the table function is invoked.

## Syntax

```

void *
FNC_TblGetCtx(void)

```

## Usage Notes

Before calling this function, the general scratchpad must be allocated by a call to FNC\_TblAllocCtx.



Do not store pointers in the scratchpad that reference other structures in the scratchpad, because the returned address of the scratchpad is not the same for subsequent invocations of the table function. Instead, use relative addressing, such as offsets from the current address of the scratchpad.

The general scratchpad is local to each copy of the table function for the current transaction or request running on each AMP vproc. It is not retained for subsequent transactions or requests.

Data changed in the scratchpad is retained for subsequent calls.

This function can only be called from within a table function. Calling this function from a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

## Example Using FNC\_TblGetCtx

```
typedef struct {
    unsigned short control_flag;
    int            qfd;
    ...
} scratchpad;

scratchpad      *options;
FNC_Phase      Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:
        {
            ...
        }
        case TBL_INIT:
        {
            options = FNC_TblGetCtx();
            if (options->control_flag)
            {
                ...
            }
        }
        ...
    }
}
...
```

## FNC\_TblGetNodeData

Returns node IDs and AMP IDs for all online AMP vprocs, allowing table functions and table operators to configure themselves to run on specific AMPs.

### Syntax

```
FNC_Node_Info_t *
FNC_TblGetNodeData(void)
```

### Return Type

A pointer to an FNC\_Node\_Info\_t structure, which lists the online AMPs.

FNC\_Node\_Info\_t is defined as:

```
typedef struct FNC_Node_Info_t {
    int NumAMPNodes;
    int NumAMPs;
    AMP_Node_t AN[1]; /* number varies with number of AMP vprocs */
                     /* (one per AMP vproc) */
} FNC_Node_Info_t;
```

Member ...	Specifies ...
<i>NumAMPNodes</i>	the total number of AMPs on the same node as the invoking AMP. If this function is invoked from a table operator that is associated with a map, then <i>NumAMPNodes</i> is the total number of AMPs on the same node within the specified map.
<i>NumAMPs</i>	the total number of AMPs that are in online or hold state within the same node as the invoking AMP. If this function is invoked from a table operator that is associated with a map, then <i>NumAMPs</i> is the total number of AMPs that are in online or hold state within the specified map and on the same node as the invoking AMP.
<i>AMP_Node_t</i>	an array listing the node IDs and AMP IDs of the online AMP vprocs. The value of <i>NumAMPs</i> determines the number of elements in the array. AMP_Node_t is defined as: <pre>typedef struct AMP_Node_t {     unsigned short NodeId;     unsigned short AMPId; } AMP_Node_t;</pre> <ul style="list-style-type: none"> <li>• NodeId specifies the unique number of the node.</li> <li>• AMPId specifies the unique number of the AMP.</li> </ul> The information is in ascending order, first by NodeId and then by AMPId.

Member ...	Specifies ...
	<p><b>Note:</b></p> <p>In the event of a node failure, a HSN (hot standby node) takes over the online AMP vprocs from the failed node. In this case, FNC_TblGetNodeData returns the node ID of the HSN instead of the failed node. Therefore, you may get results that are not in the order expected with regards to the node ID.</p>

## Usage Notes

Take the following steps to configure copies of a table function to run on specific AMPs only:

1. Call FNC\_TblGetNodeData to get the list of all node numbers and AMP vproc numbers.
2. Determine which AMP vprocs will run copies of the table function.

For example, if only one AMP vproc is supposed to participate per node, the logic can simply pick the first AMP vproc on each node in the list, because the information is returned in ascending order.

3. Call FNC\_AMPInfo to get the local node and AMP vproc number.

For table functions:

IF the local node and AMP vproc number ...	THEN ...
were selected to run a copy of the table function	participate in table function processing.
were not selected to run a copy of the table function	call FNC_TblOptOut.

For table operators, include an if statement to determine whether the local node and AMP were selected. If they were selected, the control flow goes through the processing of the operator. Otherwise, processing is skipped.

This function can only be called from within a table function or a table operator. Calling this function from a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

## Example Using FNC\_TblGetNodeData

```
FNC_Node_Info_t *NodeInfo;

NodeInfo = FNC_TblGetNodeData();
```

## FNC\_TblOpBindAttributeByNdx

Allows table operator writers to bind a specific attribute value in the current row of an output stream to a memory location.

Returns an integer indicating TBLOP\_SUCCESS or TBLOP\_ABORT.

## Syntax

```
int
FNC_TblopBindAttributeByNdx(FNC_TblopHandle_t  *handle,
                           int                 index,
                           void                *valueptr,
                           int                 null_ind,
                           int                 length);
```

### Syntax Elements

#### *handle*

the handle to the current write context.

For details about the FNC\_TblopHandle\_t structure, see [Table Operator Interface Functions](#).

#### *index*

the index of an attribute. The range of values is from 0 to *i*-1, where 0 is the index of the first attribute.

#### *valueptr*

a pointer to the memory location where the value of the attribute resides.

#### *null\_ind*

whether the attribute is null. -1 indicates null and 0 indicates not null.

#### *length*

the size in bytes of the value pointed to by *valueptr*. For character varying data types, the length includes the 2 byte length field.

## Usage Notes

This function returns an error in the following cases:

- The stream was not opened with the TBLOP\_NOOPTIONS option.
- The direction is not output.
- The stream number or index is out of bounds.
- The length is invalid for the attribute type.

This function links an attribute value to location *valueptr*. If the value referenced by *valueptr* changes between the call to `FNC_TblOpBindAttributeByNdx` and the call to `FNC_TblOpWrite`, the value written in the database is the value referenced by *valueptr* at the time of the `FNC_TblOpWrite` call.

## Example Using `FNC_TblOpBindAttributeByNdx`

See [C Table Operator](#) for an example of how to use this function.

## `FNC_TblOpBytesTransferred`

Records the number of bytes transferred between the database and the foreign server by the table operator.

### Syntax

```
void
FNC_TblOpBytesTransferred(unsigned long in,
                          unsigned long out)
```

### Syntax Elements

*in*

IN parameter

The number of bytes transferred into the database from the foreign server.

*out*

IN parameter

The number of bytes transferred from the database to the foreign server.

### Usage Notes

This routine is callable on an AMP vproc only by a table operator.

## `FNC_TblOpClose`

Allows table operator writers to close an input or output stream.

Returns an integer indicating `TBLOP_SUCCESS`, `TBLOP_ERROR`, or `TBLOP_ABORT`.

### Syntax

```
int
FNC_TblOpClose(FNC_TblOpHandle_t *handle);
```

## Syntax Elements

### *handle*

the handle associated with an input or output stream.

For details about the `FNC_TbIOpHandle_t` structure, see [Table Operator Interface Functions](#).

## Usage Notes

You can use this function to close an input or output stream within a table operator invocation. The function closes the stream. When closing an output stream in protected mode, this function flushes the buffer to the database. After calling this function, Read and Write calls are no longer valid; however, you can call `FNC_TbIOpClose` prior to reading all of the rows.

You can close and reopen an input stream within a request that did not specify a PARTITION BY clause. You can close an output stream a single time within a table operator invocation.

If you specify the PARTITION BY clause, then opening or closing a stream refers to the rows within the partition. For example, End of File occurs after the last row in the partition is read. If you do not specify the PARTITION BY clause, opening or closing a stream refers to the rows within the AMP.

The following are not allowed and result in an error:

- Close an input or output stream that is not open.
- Close and reopen an output stream.

## Example Using FNC\_TbIOpClose

See [C Table Operator](#) for an example of how to use this function.

## FNC\_TbIOpDisableCoGroup

Disables the cogroup functionality for table operators that handle multiple input streams.

## Syntax

```
void
FNC_TbIOpDisableCoGroup();
```

## Usage Notes

You can call `FNC_TbIOpDisableCoGroup` in the contract function of the table operator to turn off the cogroup functionality.

**Note:**

If cogroup is disabled, table operators that handle multiple input streams may return different results on systems with different configurations where the number of AMPs differ. To get consistent results on different configurations, cogroup must be enabled.

## FNC\_TblOpGetAsClauseName

Retrieves the alias name (AS name) associated with an input stream.

### Syntax

```
void
FNC_TblOpGetAsClauseName(int          streamno,
                          Stream_Direction_en direction,
                          char          *AsNamePtr,
                          int          buflen,
                          int          *namelen);
```

### Syntax Elements

***streamno***

the stream number.

***direction***

the direction of the stream: 'W' or 'R'.

***AsNamePtr***

the buffer to hold the AS name.

***buflen***

the length of the buffer pointed to by *AsNamePtr*.

***namelen***

the buffer to hold the actual name length.

### Usage Notes

The function returns the name specified in the AS *name* clause in UNICODE and the name length. If there is no AS name, the function returns a null buffer and sets *namelen* to 0.

## FNC\_TblOpGetAttributeByNdx

Allows table operator writers to access a specific input attribute value in the current row of a stream.

Returns an integer indicating TBLOP\_SUCCESS, TBLOP\_ERROR, or TBLOP\_ABORT.

This function will return an error in the following cases:

- The handle is associated with an output stream.
- The stream number or index is out of bounds.
- The stream is not open.
- The stream is not set up to access attributes directly. To access attributes directly, it should be opened with options TBLOP\_NOOPTIONS.

## Syntax

```
int
FNC_TblOpGetAttributeByNdx(FNC_TblOpHandle_t *handle,
                           int index,
                           void **valueptr,
                           int *null_ind,
                           int *length);
```

## Syntax Elements

### *handle*

the handle to the current read context.

For details about the FNC\_TblOpHandle\_t structure, see [Table Operator Interface Functions](#).

### *index*

the index of an attribute. The range of values is from 0 to  $i-1$ , where 0 is the index of the first attribute.

### *valueptr*

location of pointer to the attribute value to be returned.

### *null\_ind*

whether the attribute is null. -1 indicates null and 0 indicates not null.

### *length*

the size in bytes of the value pointed to by *valueptr*. For character varying data types, the length includes the 2 byte length field.



## Example Using FNC\_TblOpGetAttributeByNdx

See [C Table Operator](#) for an example of how to use this function.

## FNC\_TblOpGetBaseInfo

Get the information on the base type or attribute types for a UDT or complex data type (CDT).

### Syntax

```
void
FNC_TblOpGetBaseInfo(FNC_TblOpColumnDef_t *colDefs,
                     UDT_BaseInfo_t      *baseInfo)
```

### Syntax Elements

#### *colDefs*

A pointer to a structure containing information about the column definitions of a stream passed or returned from a table operator. The size of this structure depends on the number of attributes in the table. This is an input parameter.

#### *baseinfo*

A pointer to an array of structures, with each entry containing the metadata needed to describe a UDT or CDT column. The size of this structure depends on the number of columns in the table. If the column is a structured UDT, either with or without attributes that are structured UDT, metadata about the attributes may be retrieved by a subsequent call to FNC\_TblOpGetStructuredAttributeInfo.

This is an output parameter.

### Usage Notes

This routine is only supported with table operators and enables a UDT or complex type to be passed as an input column or returned as an output column.

This routine supports the following UDTs and CDTs:

- ARRAY/VARRAY
- DATASET
- Distinct and structured UDTs
- Geospatial types: ST\_Geometry, MBR, MBB
- JSON
- Period types
- XML

**Note:**

The base\_\* fields of the UDT\_BaseInfo\_t structure are not filled in for structured UDTs. Since structured UDTs may have many attributes and may also contain an arbitrary level of nesting, metadata about the attributes of a structured UDT is retrieved using FNC\_TblOpGetStructuredAttributeInfo. The FNC\_TblOpGetStructuredAttributeInfo routine returns an array of attribute\_info\_t structures corresponding to all of the attributes in the structured UDT.

For more information about the FNC\_TblOpColumnDef\_t and UDT\_BaseInfo\_t structures, see [Table Operator Data Structures](#).

**Example: FNC\_TblOpGetBaseInfo**

The following shows how FNC\_TblOpGetBaseInfo can be used in a contract function.

```
int tblopudt_metain_contract(
    INTEGER *Result,
    int *indicator_Result,
    char sqlstate[6],
    SQL_TEXT extname[129],
    SQL_TEXT specific_name[129],
    SQL_TEXT error_message[257])
{
    FNC_TblOpColumnDef_t    *oCols;
    FNC_TblOpColumnDef_t    *iCols;
    UDT_BaseInfo_t *udtBaseInfo;
    Stream_Fmt_en    format;
    InputInfo_t *icolinfo;
    int incount, outcount , ocolcount;
    int i,j, totalcols;
    char mycontract[] = "this is my contract... this is my contract... this is
my contract...";
    char msg[500] = "";

    FNC_TblOpGetStreamCount(&incount, &outcount);
    if(incount == 0)
    {
        SetError("U0003", "mift1 requires number of input streams to be > 0.");
        return -1;
    }

    icolinfo = FNC_malloc (incount * sizeof(InputInfo_t));

    totalcols = 0;
```

```

for(i=0; i < incount; i++)
{
    icolinfo[i].colcount = FNC_TblOpGetColCount(i, ISINPUT);
    totalcols += icolinfo[i].colcount;

    icolinfo[i].iCols = FNC_malloc(TblOpSIZECOLDEF(icolinfo[i].colcount));
    TblOpINITCOLDEF(icolinfo[i].iCols, icolinfo[i].colcount);
    FNC_TblOpGetColDef(i, ISINPUT, icolinfo[i].iCols);

    /* Retreive UDT metadata info for each column and write to trace table */
    iCols = icolinfo[i].iCols;
    icolinfo[i].baseColInfo = FNC_malloc(sizeof(UDT_BaseInfo_t) * iCols-
>num_columns);
    FNC_TblOpGetBaseInfo(iCols,icolinfo[i].baseColInfo);

}

/* Allocate space for columns. */
oCols = (FNC_TblOpColumnDef_t *)FNC_malloc( TblOpSIZECOLDEF(totalcols) );
memset(oCols, 0 , TblOpSIZECOLDEF(totalcols) );
oCols->num_columns = totalcols;
oCols->length = TblOpSIZECOLDEF(totalcols) - (2 * sizeof(int)) ;
TblOpINITCOLDEF(oCols, totalcols);
ocolcount = 0;

/* Copy input columns to output columns. */
for(j=0; j < incount; j++)
{
    iCols = icolinfo[j].iCols;
    for(i=0;i < iCols->num_columns;i++)
    {
        oCols->column_types[ocolcount].datatype = VARCHAR_DT;
        oCols->column_types[ocolcount].charset = LATIN_CT;
        oCols->column_types[ocolcount].size.length = 32;
        ocolcount++;
    }
}

FNC_TblOpSetContractDef(mycontract, strlen(mycontract)+1);
/* Define output columns. */
FNC_TblOpSetOutputColDef(0, oCols);
format = INDICFMT1;
FNC_TblOpSetFormat("RECFMT", 0, ISINPUT, &format, sizeof(format));
FNC_TblOpSetFormat("RECFMT", 0, ISOUTPUT, &format, sizeof(format));

```

```

    FNC_free(oCols);
    for(i=0; i < incount; i++)
    {
        FNC_free(icolinfo[i].iCols);
        FNC_free(icolinfo[i].baseColInfo);
    }

    FNC_free(icolinfo);
    *Result = 1;
}

```

## FNC\_TblOpGetColCount

Returns the number of columns in a stream.

### Syntax

```

int
FNC_TblOpGetColCount(int          streamno,
                     Stream_Direction_en direction);

```

### Syntax Elements

***streamno***

the stream number.

***direction***

the direction of the stream: 'W' or 'R'.

## FNC\_TblOpGetColDef

Retrieves column definitions of the stream specified by the input parameters.

### Syntax

```

void
FNC_TblOpGetColDef(int          streamno,
                   Stream_Direction_en direction,
                   FNC_TblOpColumnDef_t *coldef);

```

## Syntax Elements

### *streamno*

the stream number.

### *direction*

the direction of the stream: 'W' or 'R'.

### *coldef*

the location of the buffer where the column definitions will be placed.

For details about the `FNC_TblOpColumnDef_t` structure, see [Table Operator Data Structures](#).

## Usage Notes

If this function is not called by a table operator with parameter style `SQLTABLE`, `FNC_TblOpGetColDef` sets an error.

Before calling `FNC_TblOpGetColDef`, you should invoke the following function and macro to allocate space for the buffer where the column definitions will be returned. Get the column count once and save it for use multiple times to avoid unnecessary FNC call overhead.

```
FNC_TblOpColumnDef_t *coldef;

int ColCount = FNC_TblOpGetColCount(0, 'R');
coldef = FNC_malloc(TblOpSIZECOLDEF(ColCount));
```

Function `FNC_TblOpGetColCount` retrieves the number of columns in an input stream and `TblOpSIZECOLDEF` computes the required size in bytes to store the column definitions of the input stream.

After memory is allocated, use the following macro to initialize the data structure:

```
TblOpINITCOLDEF(coldef, ColCount);
```

After initialization, function `FNC_TblOpGetColDef` can be called:

```
FNC_TblOpGetColDef(0, 'R', coldef);
```

Once the table operator is done with the column definitions in *coldef*, you need to release the memory:

```
FNC_free(coldef);
```

## Example Using FNC\_TblOpGetColDef

See [C Table Operator](#) for an example of how to use this function.

## FNC\_TblOpGetContractDef

Retrieves the contract function context.

### Syntax

```
void
FNC_TblOpGetContractDef(void *ContractBuf,
                        int    inContractLen,
                        int    *outContractLen);
```

### Syntax Elements

#### ***ContractBuf***

a pointer to a buffer in which the contract context will be copied.

#### ***inContractLen***

the number of bytes allocated for the *ContractBuf* buffer.

#### ***outContractLen***

the size of the contract function context.

### Usage Notes

The buffer where the contract context will be copied needs to be allocated by the contract function or table operator before calling FNC\_TblOpGetContractDef. You can call FNC\_TblOpGetContractLength to obtain the length of the context. This length can be passed to FNC\_malloc to allocate the space needed. You must release the space once you are done with the context:

```
ctxPtr = FNC_malloc(FNC_TblOpGetContractLength());
...
FNC_free(ctxPtr);
```

## FNC\_TblOpGetContractLength

Retrieves the length of the contract function context.

## Syntax

```
int
FNC_TblOpGetContractLength()
```

## FNC\_TblOpGetContractPhase

Returns the phase in the parser from which the contract function is being called:

Parser Phase	Meaning
FNC_CTRCT_GET_ALLCOLS_PHASE = 0	Function returns all remote table columns.
FNC_CTRCT_VALIDATE_PHASE = 1	Inputs are correct. Contract function can be called multiple times from this phase. <b>Note:</b> This phase is not used.
FNC_CTRCT_COMPLETE_PHASE = 2	Last call of contract function. Any foreign server actions that must be done must be completed.
FNC_CTRCT_DDL_PHASE = 3	Execution of CREATE SERVER statement is being completed. Connectivity must be verified.
FNC_CTRCT_DEFINE_SERVER_PHASE = 4	CREATE VIEW or CREATE MACRO statement is being executed. Custom clause data may be invalid.

## Syntax

```
int
FNC_TblOpGetContractPhase()
```

## Usage Notes

This routine is callable on a PE vproc only by a table operator.

## FNC\_TblOpGetCountHashByDef

Returns the number of columns in a HASH BY clause associated with an input stream.

## Syntax

```
short
FNC_TblOpGetCountHashByDef(int streamno);
```

## Syntax Elements

***streamno***

an input stream number.

## FNC\_TblOpGetCountLocalOrderByDef

Returns the number of columns in a LOCAL ORDER BY clause associated with an input stream.

## Syntax

```
short
FNC_TblOpGetCountLocalOrderByDef(int  streamno);
```

## Syntax Elements

***streamno***

an input stream number.

## FNC\_TblOpGetCustomKeyCount

Returns the number of Custom clause keys for a table operator.

## Syntax

```
int
FNC_TblOpGetCustomKeyCount()
```

## Example Using FNC\_TblOpGetCustomKeyCount

The following query has 2 Custom clause keys: SUM\_AGG and AVG\_AGG.

FNC\_TblOpGetCustomKeyCount returns 2 when called from the table operator or contract function associated with this query.

```
SELECT *
FROM udaggregation (
    on (select * from tab1)
    using SUM_AGG('A', 'B')
        AVG_AGG('C')
) as D;
```



## Related Information

For details about the Custom clause of a table operator, see information about the SELECT statement FROM clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## FNC\_TblOpGetCustomKeyInfoAt

### Syntax

```
int
FNC_TblOpGetCustomKeyInfoAt(int          index,
                             Key_info_t *out_info);
```

### Syntax Elements

#### *index*

the index to a key-value pair.

#### *out\_info*

the location of the data structure where the number of values, total size, type, key, and the size of the key will be returned.

For details about the Key\_info\_t structure, see [Table Operator Data Structures](#).

## Usage Notes

Keys are indexed in the order that they appear in the query, starting at 0. In the following query, SUM\_AGG would have index 0 and AVG\_AGG would have index 1.

```
SELECT *
FROM udaggregation (
    on (select * from tab1)
    using SUM_AGG('A', 'B')
         AVG_AGG('C')
) as D;
```

Before calling FNC\_TblOpGetCustomKeyInfoAt, you must allocate memory for the *out\_info* data structure:

```
out_info = FNC_malloc(sizeof(Key_info_t))
```

You must release the data structure when it is no longer needed:

```
FNC_free(out_info);
```

## Related Information

For details about the Custom clause of a table operator, see information about the SELECT statement FROM clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## FNC\_TblOpGetCustomKeyInfoOf

Retrieves the number of values associated with a key in a Custom clause, their total size in bytes, their type, key, and the size of the key.

## Syntax

```
int
FNC_TblOpGetCustomKeyInfoOf(void      *key,
                             Key_info_t *out_info);
```

### Syntax Elements

#### *key*

the location of the key.

#### *out\_info*

the location of the data structure where the number of values, total size, type, key, and the size of the key will be returned.

For details about the Key\_info\_t structure, see [Table Operator Interface Functions](#).

## Usage Notes

Before calling FNC\_TblOpGetCustomKeyInfoOf, you must allocate memory for the *out\_info* data structure:

```
out_info = FNC_malloc(sizeof(Key_info_t))
```

You must release the data structure when it is no longer needed:

```
FNC_free(out_info);
```

## Related Information

For details about the Custom clause of a table operator, see information about the SELECT statement FROM clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## FNC\_TblOpGetCustomValuesOf

Retrieves all the values associated with a given key in a Custom clause and their corresponding lengths.

The values associated with a given key in a Custom clause and the length of each value.

The value that is returned can be an OID (object identifier). The OID is in the form of a client locator.

### Syntax

```
int
FNC_TblOpGetCustomValuesOf(Key_info_t *key);
```

### Syntax Elements

#### *key*

the location of the data structure where *key* is specified and where values and lengths will be placed.

For details about the Key\_info\_t structure, see [Table Operator Data Structures](#).

### Usage Notes

Before calling FNC\_TblOpGetCustomValuesOf, you must allocate memory for the values to be returned:

```
key->values_r = FNC_malloc(sizeof(values_t)*key->numOfVal);
```

You must release the memory when the values are no longer needed:

```
FNC_free(key->values_r);
```

### Example Using FNC\_TblOpGetCustomValuesOf

See [C Table Operator](#) for an example of how to use this function.

### Related Information

For details about the Custom clause of a table operator, see information about the SELECT statement FROM clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## FNC\_TblOpGetExternalQuery

Generates the text query string for the foreign server and returns the interface version that is currently supported.

### Syntax

```
void
FNC_TblOpGetExternalQuery(FNC_TblOpColumnDef_t *colDefs,
                          ServerType           serverType,
                          ExtOpSetType         opSet,
                          int                   *interfaceVersion,
                          unsigned char         **extQryPtr,
                          unsigned int          *extQryLenPtr)
```

### Syntax Elements

#### *colDefs*

IN parameter

A list of column definitions that may occur in a WHERE clause by the foreign server.

#### *serverType*

IN parameter

ServerType is defined as follows:

```
typedef enum
{
    ANSISQL = 1,
    HADOOP = 2
} serverType_et;
```

```
typedef int ServerType;
```

- If ANSI SQL, the entire subquery for the table of the foreign server is returned.
- If HADOOP, only the WHERE clause portion of the query is returned.

#### *opSet*

IN parameter

A set of valid operators supported on the foreign server.

ExOpSetType is defined as follows:

```

typedef enum
{
    Eq_ET,
    Ne_ET,
    Gt_ET,
    Le_ET,
    Lt_ET,
    And_ET,
    Or_ET,
    Not_ET,
    Between_ET,
    In_ET,
    NotIn_ET,
    Ge_ET,
    Like_ET
} extoptype_et;

typedef BYTE ExtOpType;
typedef unsigned int ExtOpSet;

typedef struct ExtOpSetType {
    ExtOpSet ExtOpSetList;
} ExtOpSetType;

```

***interfaceVersion***

IN/OUT parameter

A pointer to the interface version:

- The caller passes in the desired interface version as the argument.
- The routine returns the actual interface version that is currently returns the actual interface version that is currently

***extQryLenPtr***

OUT parameter

A pointer to the generated text query string for the foreign server. The query string is null-terminated.

***extQryLenPtr***

OUT parameter

A pointer to the length of the external query (in bytes).

## Usage Notes

This routine is callable on a PE vproc only by a table operator.

### Note:

FNC\_TblOpGetExternalQuery calls FNC\_malloc to allocate memory for the buffer specified by \*extQryPtr. Unless the routine returns \*extQryPtr as NULL, you must use FNC\_free to free the allocated memory after processing the data.

## FNC\_TblOpGetFormat

Returns the default format or the format set in the contract function by a previous call to FNC\_TblOpSetFormat.

### Syntax

```
void
FNC_TblOpGetFormat(char          *attribute,
                    int           streamno,
                    Stream_Direction_en direction,
                    void          *outValue,
                    int           outSize);
```

### Syntax Elements

#### *attribute*

the format attribute to be retrieved. Currently, the only possible value is "RECFMT".

#### *streamno*

the stream number.

#### *direction*

the stream direction: 'R' or 'W'.

#### *outValue*

the buffer where the format attribute value is returned.

#### *outSize*

the size in bytes of the *outValue* buffer.

## Usage Notes

Format attribute "RECFMT" defines the input/output record format and has the following two possible values:

```
typedef enum
{
    INDICFMT1 = 1,          /* IndicData with row separator sentinels */
    INDICBUFFMT1 = 2       /* IndicData with NO row or partition separator
sentinels */
} Stream_Fmt_en;
```

When format attribute is "RECFMT", buffer *outvalue* should be of size `sizeof(Stream_Fmt_en)` and the attribute value returned would be INDICFMT1 (IndicData with row separator sentinels) or INDICBUFFMT1 (IndicData with no row or partition separator sentinels). Memory must be allocated for buffer *outvalue* before FNC\_TblOpGetFormat is called and released after the buffer is no longer needed.

## FNC\_TblOpGetFunctionDef

Provides access to information about the calling table operator.

### Syntax

```
void
FNC_TblOpGetFunctionDef(char          *fnameBuff,
                        int            fnameBuff_len,
                        BYTE            *execmode,
                        unsigned short int *chatype,
                        boolean_t       *casesensitivity);
```

### Syntax Elements

#### ***fnameBuff***

a pointer to a buffer where the name of the table operator is returned.

#### ***fnameBuff\_len***

the length of the *fnameBuff* buffer.

#### ***execmode***

a pointer to the memory location where the execution mode of the table operator is returned.

***chartype***

a pointer to the memory location where the character set of the table operator is returned.

***casesensitivity***

a pointer to the memory location where the case sensitivity of the table operator is returned.

**Usage Notes**

You must allocate memory for the table operator name, execution mode, character set, and case sensitivity before calling `FNC_TblOpGetFunctionDef`. You must release the memory when this information is no longer needed.

## FNC\_TblOpGetHashByDef

Retrieves the HASH BY clause information of the input stream for a table operator.

**Syntax**

```
void
FNC_TblOpGetHashByDef(int      streamno,
                      FNC_Names_t *out_colname);
```

**Syntax Elements*****streamno***

the input stream number.

***out\_colname***

a pointer to the location where the column names will be stored.

For details about the `FNC_Names_t` structure, see [Table Operator Data Structures](#).

**Usage Notes**

Before calling `FNC_TblOpGetHashByDef`, you must allocate memory for the *out\_colname* buffer:

```
FNC_Names_t *out_colname = FNC_malloc(TblOpSIZEHASHBY(
                                FNC_TblOpCountHashByDef(streamno)) );
```

You must release the allocated memory when *out\_colname* is no longer needed. If you used `FNC_malloc` to allocate the memory, use `FNC_free` to release it.



## FNC\_TblOpGetInnerContract

Gets the contract definition of a nested inner table operator for the outer table operator to use.

### Syntax

```
void
FNC_TblOpGetInnerContract(void **innerContract,
                          int   *contractLen)
```

### Syntax Elements

#### *innerContract*

IN/OUT parameter

Input argument: Identifies the buffer which will hold the contract definition information.

Return value:

- The contract definition of the inner table operator.
- NULL, if the inner contract function does not exist.

#### *contractLen*

OUT parameter

The length of the contract definition.

### Usage Notes

This routine is callable on a PE vproc only by a table operator.

#### Note:

FNC\_TblOpGetInnerContract calls FNC\_malloc to allocate memory for the buffer specified by *\*innerContract*. Unless the routine returns *\*innerContract* as NULL, you must use FNC\_free to free the allocated memory after processing the data.

## FNC\_TblOpGetLocalOrderByDef

Retrieves the LOCAL ORDER BY clause information of the input stream for a table operator.

### Syntax

```
void
FNC_TblOpGetLocalOrderByDef(int          streamno,
                             FNC_Names_Ord_t  *out_colname);
```

## Syntax Elements

***streamno***

an input stream number.

***out\_colname***

a pointer to the location where the column names and their ordering will be stored.

For details about the `FNC_Names_Ord_t` structure, see [Table Operator Data Structures](#).

## Usage Notes

Before calling `FNC_TblOpGetLocalOrderByDef`, you must allocate memory for the *out\_colname* buffer:

```
FNC_Names_Ord_t *out_colname = FNC_malloc( TblOpSIZEORDERBY(
                                         FNC_TblOpCountHashByDef(streamno)) );
```

You must release the allocated memory when *out\_colname* is no longer needed. If `FNC_malloc` was used to allocate the memory, use `FNC_free` to release it.

## FNC\_TblOpGetStreamCount

Gets the number of input and output streams passed to the table operator.

### Syntax

```
void
FNC_TblOpGetStreamCount(int *in_count,
                       int *out_count)
```

## Syntax Elements

***in\_count***

a pointer to an integer where the number of input streams is placed.

***out\_count***

a pointer to an integer where the number of output streams is placed.

## Usage Notes

The first input stream corresponds to stream number 0. Table operators support multiple input streams, but only one output stream.

## FNC\_TblOpGetStructuredAttributeInfo

Get the information on all of the attributes of a particular structured UDT type.

### Syntax

```
void
FNC_TblOpGetStructuredAttributeInfo(unsigned char    *udtName,
                                   int               bufSize,
                                   attribute_info_t *attributeArray)
```

### Syntax Elements

#### *udtName*

A pointer to the structured UDT type name.

#### *bufSize*

The size in bytes that was allocated to the *attributeArray* argument.

#### *attributeArray*

An array of one or more `attribute_info_t` structures that describe all of the attributes in the structured UDT. For more information about the `attribute_info_t` structure, see [Table Operator Data Structures](#).

### Usage Notes

This routine is only supported with table operators.

## Example: FNC\_TblOpGetUDTMetadata and FNC\_TblOpGetStructuredAttributeInfo

The following shows how `FNC_TblOpGetUDTMetadata` and `FNC_TblOpGetStructuredAttributeInfo` can be used in a contract function.

```
int tblopudt_contract(
    INTEGER *Result,
    int *indicator_Result,
    char sqlstate[6],
    SQL_TEXT extname[129],
    SQL_TEXT specific_name[129],
    SQL_TEXT error_message[257])
{
    FNC_TblOpColumnDef_t    *oCols;
    FNC_TblOpColumnDef_t    *iCols;
```

```

UDT_BaseInfo_t *udtBaseInfo;
Stream_Fmt_en  format;
InputInfo_t *icolinfo;
int incount, outcount , ocolcount;
int i,j, totalcols;
char mycontract[] = "this is my contract... this is my contract... this is
my contract...";
char msg[500];

FNC_TblOpGetStreamCount(&incount, &outcount);
if(incount == 0)
{
    SetError("U0003", "mift1 requires number of input streams to be > 0.");
    return -1;
}

icolinfo = FNC_malloc (incount * sizeof(InputInfo_t));

totalcols = 0;
for(i=0; i < incount; i++)
{
    icolinfo[i].colcount = FNC_TblOpGetColCount(i, ISINPUT);
    totalcols += icolinfo[i].colcount;

    icolinfo[i].iCols = FNC_malloc(TblOpSIZECOLDEF(icolinfo[i].colcount));
    TblOpINITCOLDEF(icolinfo[i].iCols, icolinfo[i].colcount);
    FNC_TblOpGetColDef(i, ISINPUT, icolinfo[i].iCols);

    /* Retreive UDT metadata info for each column and write to trace table */
    icols = icolinfo[i].iCols;
    icolinfo[i].baseColInfo = FNC_malloc(sizeof(UDT_BaseInfo_t) * icols-
>num_columns);

    FNC_TblOpGetUDTMetadata(i, ISINPUT, -1,
                           sizeof(UDT_BaseInfo_t) * icols->num_columns,
                           &(icolinfo[i].baseColInfo[0]));

    for(j=0; j < icols->num_columns; j++)
    {
        UDT_BaseInfo_t* udtbaseinfo;
        int bufferSize = 0;
        attribute_info_eon_t * attrInfo = NULL;
        char udtname[100];
        udtbaseinfo = &(icolinfo[i].baseColInfo[j]);
    }
}

```

```

        int k;
        if(udtbaseinfo->udt_indicator == 2) // Structured UDT
        {
            bufferSize = udtbaseinfo->struct_num_attributes
* sizeof(attribute_info_eon_t);
            attrInfo = FNC_malloc(bufferSize);
            memset(attrInfo,0,bufferSize);
            FNC_TblOpGetStructuredAttributeInfo(udtbaseinfo->udt_name,
                                                bufferSize,attrInfo);

            FNC_free(attrInfo);
        }
    }

}
/* Allocate space for columns. */
oCols = (FNC_TblOpColumnDef_t *)FNC_malloc( TblOpSIZECOLDEF(totalcols) );
memset(oCols, 0 , TblOpSIZECOLDEF(totalcols) );
oCols->num_columns = totalcols;
oCols->length = TblOpSIZECOLDEF(totalcols) - (2 * sizeof(int)) ;
TblOpINITCOLDEF(oCols, totalcols);
ocolcount = 0;

/* Copy input columns to output columns. */
for(j=0; j < incount; j++)
{
    iCols = icolinfo[j].iCols;
    for(i=0;i < iCols->num_columns;i++)
    {
        switch (iCols->column_types[i].datatype)
        {
            case DECIMAL1_DT:
            case DECIMAL4_DT:
            case DECIMAL8_DT:
                oCols->column_types[ocolcount].size.range.totaldigit = iCols-
>column_types[i].size.range.totaldigit;
                oCols->column_types[ocolcount].size.range.fracdigit = iCols-
>column_types[i].size.range.fracdigit;
                break;
            case DECIMAL2_DT:
                oCols-
>column_types[ocolcount].size.range.totaldigit = 5;
                oCols-
>column_types[ocolcount].size.range.fracdigit = iCols-
>column_types[i].size.range.fracdigit;

```

```

        break;
        case TIME_DT:
        case TIMESTAMP_DT:
        case PERIOD_DT:
            oCols->column_types[ocolcount].size.precision = iCols-
>column_types[i].size.precision;
            break;
        case INTERVAL_YEAR_DT:
        case INTERVAL_YTM_DT:
        case INTERVAL_MONTH_DT:
        case INTERVAL_DAY_DT:
        case INTERVAL_DTH_DT:
        case INTERVAL_DTM_DT:
        case INTERVAL_DTS_DT:
        case INTERVAL_HOUR_DT:
        case INTERVAL_HTM_DT:
        case INTERVAL HTS_DT:
        case INTERVAL_MINUTE_DT:
        case INTERVAL_MTS_DT:
        case INTERVAL_SECOND_DT:
            oCols->column_types[ocolcount].size.intervalrange = iCols-
>column_types[i].size.intervalrange;
            break;
        case ARRAY_DT:
        case UDT_DT:
        case MBB_DT:
        case MBR_DT:
            strcpy(oCols->column_types[ocolcount].udt_type,iCols-
>column_types[i].udt_type);
            oCols->column_types[ocolcount].size.length = iCols-
>column_types[i].size.length;
            break;
        default:
            oCols->column_types[ocolcount].size.length = iCols-
>column_types[i].size.length;
            break;
    }
    oCols->column_types[ocolcount].datatype = iCols-
>column_types[i].datatype;
    oCols->column_types[ocolcount].charset = iCols-
>column_types[i].charset;
    oCols->column_types[ocolcount].period_et = iCols-
>column_types[i].period_et;
    oCols->column_types[ocolcount].bytesize = iCols-

```

```

>column_types[i].bytesize;
        oCols->column_types[ocolcount].JSONStorageFormat = iCols-
>column_types[i].JSONStorageFormat;
        ocolcount++;

    }

}

FNC_TblOpSetContractDef(mycontract, strlen(mycontract)+1);
/* Define output columns. */
FNC_TblOpSetOutputColDef(0, oCols);
format = INDICFMT1;
FNC_TblOpSetFormat("RECFMT", 0, ISINPUT, &format, sizeof(format));
FNC_TblOpSetFormat("RECFMT", 0, ISOUTPUT, &format, sizeof(format));
FNC_free(oCols);
for(i=0; i < incount; i++)
{
    FNC_free(icolinfo[i].iCols);
    FNC_free(icolinfo[i].baseColInfo);
}

FNC_free(icolinfo);
*Result = 1;
}

```

## FNC\_TblOpGetUDTMetadata

Returns metadata information about one or more UDT columns for an input or output stream.

### Syntax

```

void
FNC_TblOpGetUDTMetadata(int          streamno,
                        Stream_Direction_en direction,
                        int          columnno,
                        int          bufSize,
                        UDT_BaseInfo_t *udtbaseInfo);

```

### Syntax Elements

#### *streamno*

The input stream number.

***direction***

Input or output.

***columnno***

The column number for which UDT metadata is requested. If *columnno* is -1, UDT metadata is returned for all columns.

***int bufSize***

The size of the *udtbaseInfo* buffer passed to this function.

***udtbaseInfo***

A pointer to an array of UDT\_BaseInfo\_t structures that will contain the metadata information returned by this function. If you request metadata for all input columns, you must allocate sufficient memory for all the columns.

**Usage Notes**

FNC\_TbLOpGetUDTMetadata returns an array of UDT\_BaseInfo\_t structures that contain UDT metadata information for one or more input or output UDT columns for an input stream. For nonUDT columns, the *udt\_indicator* is set to 0.

This routine is only supported with table operators. The function can be invoked in both the contract function as well as the operator. When invoked in the contract function, it can only be used with an input stream. When invoked in the operator, it can be used with either the input or output stream.

For more information about the UDT\_BaseInfo\_t structure and the *udt\_indicator*, see [Table Operator Data Structures](#).

## Example: FNC\_TbLOpGetUDTMetadata and FNC\_TbLOpGetStructuredAttributeInfo

The following shows how FNC\_TbLOpGetUDTMetadata and FNC\_TbLOpGetStructuredAttributeInfo can be used in a contract function.

```
int tblopudt_contract(
    INTEGER *Result,
    int *indicator_Result,
    char sqlstate[6],
    SQL_TEXT extname[129],
    SQL_TEXT specific_name[129],
    SQL_TEXT error_message[257])
{
    FNC_TbLOpColumnDef_t    *oCols;
    FNC_TbLOpColumnDef_t    *iCols;
```



```

UDT_BaseInfo_t *udtBaseInfo;
Stream_Fmt_en   format;
InputInfo_t *icolinfo;
int incount, outcount , ocolcount;
int i,j, totalcols;
char mycontract[] = "this is my contract... this is my contract... this is
my contract...";
char msg[500];

FNC_TblOpGetStreamCount(&incount, &outcount);
if(incount == 0)
{
    SetError("U0003", "mift1 requires number of input streams to be > 0.");
    return -1;
}

icolinfo = FNC_malloc (incount * sizeof(InputInfo_t));

totalcols = 0;
for(i=0; i < incount; i++)
{
    icolinfo[i].colcount = FNC_TblOpGetColCount(i, ISINPUT);
    totalcols += icolinfo[i].colcount;

    icolinfo[i].iCols = FNC_malloc(TblOpSIZECOLDEF(icolinfo[i].colcount));
    TblOpINITCOLDEF(icolinfo[i].iCols, icolinfo[i].colcount);
    FNC_TblOpGetColDef(i, ISINPUT, icolinfo[i].iCols);

    /* Retreive UDT metadata info for each column and write to trace table */
    icols = icolinfo[i].iCols;
    icolinfo[i].baseColInfo = FNC_malloc(sizeof(UDT_BaseInfo_t) * icols-
>num_columns);

    FNC_TblOpGetUDTMetadata(i, ISINPUT, -1,
                           sizeof(UDT_BaseInfo_t) * icols->num_columns,
                           &(icolinfo[i].baseColInfo[0]));

    for(j=0; j < icols->num_columns; j++)
    {
        UDT_BaseInfo_t* udtbaseinfo;
        int bufferSize = 0;
        attribute_info_eon_t * attrInfo = NULL;
        char udtname[100];
        udtbaseinfo = &(icolinfo[i].baseColInfo[j]);
    }
}

```

```

        int k;
        if(udtbaseinfo->udt_indicator == 2) // Structured UDT
        {
            bufferSize = udtbaseinfo->struct_num_attributes
* sizeof(attribute_info_eon_t);
            attrInfo = FNC_malloc(bufferSize);
            memset(attrInfo,0,bufferSize);
            FNC_TblOpGetStructuredAttributeInfo(udtbaseinfo->udt_name,
                                                bufferSize,attrInfo);
            FNC_free(attrInfo);
        }
    }

}

/* Allocate space for columns. */
oCols = (FNC_TblOpColumnDef_t *)FNC_malloc( TblOpSIZECOLDEF(totalcols) );
memset(oCols, 0 , TblOpSIZECOLDEF(totalcols) );
oCols->num_columns = totalcols;
oCols->length = TblOpSIZECOLDEF(totalcols) - (2 * sizeof(int)) ;
TblOpINITCOLDEF(oCols, totalcols);
ocolcount = 0;

/* Copy input columns to output columns. */
for(j=0; j < incount; j++)
{
    iCols = icolinfo[j].iCols;
    for(i=0;i < iCols->num_columns;i++)
    {
        switch (iCols->column_types[i].datatype)
        {
            case DECIMAL1_DT:
            case DECIMAL4_DT:
            case DECIMAL8_DT:
                oCols->column_types[ocolcount].size.range.totaldigit = iCols-
>column_types[i].size.range.totaldigit;
                oCols->column_types[ocolcount].size.range.fracdigit = iCols-
>column_types[i].size.range.fracdigit;
                break;
            case DECIMAL2_DT:
                oCols-
>column_types[ocolcount].size.range.totaldigit = 5;
                oCols-
>column_types[ocolcount].size.range.fracdigit = iCols-
>column_types[i].size.range.fracdigit;

```

```

        break;
        case TIME_DT:
        case TIMESTAMP_DT:
        case PERIOD_DT:
            oCols->column_types[ocolcount].size.precision = iCols-
>column_types[i].size.precision;
            break;
        case INTERVAL_YEAR_DT:
        case INTERVAL_YTM_DT:
        case INTERVAL_MONTH_DT:
        case INTERVAL_DAY_DT:
        case INTERVAL_DTH_DT:
        case INTERVAL_DTM_DT:
        case INTERVAL_DTS_DT:
        case INTERVAL_HOUR_DT:
        case INTERVAL_HTM_DT:
        case INTERVAL HTS_DT:
        case INTERVAL_MINUTE_DT:
        case INTERVAL_MTS_DT:
        case INTERVAL_SECOND_DT:
            oCols->column_types[ocolcount].size.intervalrange = iCols-
>column_types[i].size.intervalrange;
            break;
        case ARRAY_DT:
        case UDT_DT:
        case MBB_DT:
        case MBR_DT:
            strcpy(oCols->column_types[ocolcount].udt_type,iCols-
>column_types[i].udt_type);
            oCols->column_types[ocolcount].size.length = iCols-
>column_types[i].size.length;
            break;
        default:
            oCols->column_types[ocolcount].size.length = iCols-
>column_types[i].size.length;
            break;
    }
    oCols->column_types[ocolcount].datatype = iCols-
>column_types[i].datatype;
    oCols->column_types[ocolcount].charset = iCols-
>column_types[i].charset;
    oCols->column_types[ocolcount].period_et = iCols-
>column_types[i].period_et;
    oCols->column_types[ocolcount].bytesize = iCols-

```

```

>column_types[i].bytesize;
    oCols->column_types[ocolcount].JSONStorageFormat = iCols-
>column_types[i].JSONStorageFormat;
    ocolcount++;

    }

}

FNC_TblOpSetContractDef(mycontract, strlen(mycontract)+1);
/* Define output columns. */
FNC_TblOpSetOutputColDef(0, oCols);
format = INDICFMT1;
FNC_TblOpSetFormat("RECFMT", 0, ISINPUT, &format, sizeof(format));
FNC_TblOpSetFormat("RECFMT", 0, ISOUTPUT, &format, sizeof(format));
FNC_free(oCols);
for(i=0; i < incount; i++)
{
    FNC_free(icolinfo[i].iCols);
    FNC_free(icolinfo[i].baseColInfo);
}

FNC_free(icolinfo);
*Result = 1;
}

```

## FNC\_TblOpGetUniqID

Returns the unique identifier associated with a table operator.

### Syntax

```

int
FNC_TblOpGetUniqID()

```

### Usage Notes

When multiple table operators are present in a single query, as in a map reduce operation such as the following, you can use the unique table identifier to distinguish between each of the table operators:

```

SEL * FROM MyOperator(ON (SEL * FROM MyOperator ON (tab1) d2)) d1;

```

## FNC\_TblOpIsDimension

Returns 1 (TRUE) if the input to the table operator is DIMENSION, 0 (FALSE) otherwise.

### Syntax

```
int
FNC_TblOpIsDimension(int          streamno,
                     Stream_Direction_en direction);
```

### Syntax Elements

***streamno***

the stream number.

***direction***

the direction of the stream: 'W' or 'R'.

## FNC\_TblOpOpen

Allows table operator writers to open an input or output stream.

Returns a pointer to an FNC\_TblOpHandle\_t structure or NULL if an error occurred.

For details about the FNC\_TblOpHandle\_t structure, see [Table Operator Data Structures](#).

### Syntax

```
FNC_TblOpHandle_t *
FNC_TblOpOpen(int          streamno,
               char         mode,
               unsigned int options);
```

### Syntax Elements

***streamno***

the stream number.

***mode***

the mode of the stream : 'R' or 'W'.

***options***

whether individual attributes can be accessed or modified. Valid values are:

- **TBLOP\_NOOPTIONS** - Indicates that the attributes can be accessed or modified individually via two options:
  - Using `FNC_TblOpGetAttributeByNdx` or `FNC_TblOpBindAttributeByNdx`.
  - Directly setting indicators, column pointers and lengths of the row in the handle.
- **TBLOP\_RAWMODE** - Indicates that the access mode will be raw row. The raw row length and the body of the raw row can be accessed and set. For details, see [FNC\\_TblOpRead](#) and [FNC\\_TblOpWrite](#).

LOB columns are not supported when **TBLOP\_RAWMODE** is selected for *options* in either of the input or output streams.

## Usage Notes

You can use this function to open an input stream multiple times within a table operator invocation. The function initializes the specified stream number *streamno* iterator interface for reading or writing, based on the *mode* value. Each open resets the read position to the start of the stream. You can decide to never open the input stream or to open without reading the input stream.

You can open an output stream a single time within a table operator invocation. You can decide to never open the output stream or open without writing the output stream.

If you specify the **PARTITION BY** clause, then opening a stream refers to the rows within the partition. For example, End of File occurs after the last row in the partition is read. If you do not specify the **PARTITION BY** clause, opening a stream refers to the rows within the AMP.

The following are not allowed and result in an error:

- Open an input or output stream that is already open.
- Close and reopen an output stream.

## Example Using `FNC_TblOpOpen`

See [C Table Operator](#) for an example of how to use this function.

## `FNC_TblOpRead`

Allows table operator writers to read rows from an input stream. The function sets the read context to the next input row of data and returns an integer that indicates **TBLOP\_SUCCESS**, **TBLOP\_EOF** (no more data), **TBLOP\_ABORT**, or **TBLOP\_ERROR**.

## Syntax

```
int
FNC_TblOpRead(FNC_TblOpHandle_t *handle)
```

## Syntax Elements

### *handle*

a handle associated with an input stream. This should be the handle returned by FNC\_TblOpOpen.

For details about the FNC\_TblOpHandle\_t structure, see [Table Operator Data Structures](#).

## Usage Notes

You can read all or a subset of the input rows. For more efficient access, the data can be directly accessed through the handle structure.

You must call FNC\_TblOpOpen first, then pass the handle returned from FNC\_TblOpOpen as an argument to FNC\_TblOpRead. If you call FNC\_TblOpOpen with the *options* field set to TBLOP\_NOOPTIONS, the read call prepares the field access allowing direct access to individual attributes via FNC\_TblOpGetAttributeByNdx or directly via handle->row->indicators, columnptr[index] and lengths[index].

If the *options* field is set to TBLOP\_RAWMODE, the raw row can be accessed as follows:

```
Access handle->row to access the current raw row length,  
IndicData format, record body)
```

TBLOP\_EOF is returned if you read past the End of File. If you specify the PARTITION BY clause, End of File occurs after the last row in the partition is read.

### Example Using FNC\_TblOpRead

See [C Table Operator](#) for an example of how to use this function.

### Related Information

For details about the Custom clause of a table operator, see information about the SELECT statement FROM clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

For information about the current\_row\_t structure which stores information about the current row, see [Table Operator Data Structures](#).

## FNC\_TblOpReadBuf

Provides access to the current input buffer.

Returns an integer that indicates TBLOP\_SUCCESS, TBLOP\_EOF (no more data), TBLOP\_ABORT, or TBLOP\_ERROR.

**Syntax**

```
int
FNC_TblOpReadBuf(BYTE **bufptr,
                  int    *length)
```

**Syntax Elements*****bufptr***

a pointer to the current input buffer.

***length***

a pointer to the length in bytes of the input buffer.

**Usage Notes**

This function returns a pointer to the current input buffer. It is used for high performance direct buffer read access and requires that the buffer be in a standard row format. The buffer format is IndicData.

Use [FNC\\_TblOpReadBufEx](#) instead of this routine if you need support for handling multiple input streams.

**FNC\_TblOpReadBufEx**

Used for reading a data buffer and supports multiple input streams.

Returns an integer that indicates TBLOP\_SUCCESS, TBLOP\_EOF (no more data), TBLOP\_ABORT, or TBLOP\_ERROR.

**Syntax**

```
int
FNC_TblOpReadBufEx(FNC_TblOpHandle_t  *handle,
                   BYTE                **bufptr,
                   int                  *length)
```

**Syntax Elements*****handle***

the handle associated with an input stream.

For details about the FNC\_TblOpHandle\_t structure, see [Table Operator Data Structures](#).

***bufptr***

a pointer to the location of the data buffer.



***length***

a pointer to the actual length in bytes of the buffer specified by *bufptr*.

**Usage Notes**

`FNC_TblOpReadBufEx` is similar to `FNC_TblOpReadBuf`, except it takes an input stream handle as an argument so that it can handle multiple input streams.

The function is used for high performance direct buffer read access and requires that the buffer be in a standard row format. The buffer format is `IndicData`.

If the buffer is empty, the call blocks after requesting the database to fill the buffer with more data.

**FNC\_TblOpSetContractDef**

Sets an opaque binary string value that the contract function passes to the associated table operator at execution time. This string is referred to as the contract function context.

**Syntax**

```
void
FNC_TblOpSetContractDef(void *ContractBuf,
                        int   BufSize)
```

**Syntax Elements*****ContractBuf***

the location of a binary string representing the contract function context.

***BufSize***

the size of the binary string.

**Usage Notes**

The length of the passed opaque string is limited to a maximum of 64K bytes. This function is only valid if called within the contract function.

**FNC\_TblOpSetDisplayLength**

Resets the lengths in column definitions for `VARCHAR` data types.

## Syntax

```
void
FNC_TblOpSetDisplayLength(Stream_Direction_en    direction,
                          FNC_TblOpColumnDef_t *colDefs)
```

## Syntax Elements

### *direction*

IN parameter

Stream\_Direction\_en is defined as follows:

```
typedef enum
{
    ISOUTPUT = 'W',
    ISINPUT  = 'R'
} Stream_Direction_en;
```

Specify the input value ISINPUT for export and the value ISOUTPUT for import.

### *colDefs*

IN/OUT parameter

A pointer to the column definitions which will be returned with the modified display lengths.

## Usage Notes

This routine can be invoked for both import and export operations.

The routine is callable on a PE vproc only by a table operator.

## FNC\_TblOpSetError

Sets the SQLSTATE and detailed error message information allowing table operator writers to specify an error code and message to be displayed to the user.

## Syntax

```
void
FNC_TblOpSetError(char *SQLState,
                  char *Message)
```

## Syntax Elements

### **SQLState**

an SQLSTATE error code.

### **Message**

an error message.

## FNC\_TblOpSetExplainText

Sets the EXPLAIN text when the table operator has the hexplain custom clause set.

## Syntax

```
void
FNC_TblOpSetExplainText(int      numOfTexts,
                        char **arrayOfTexts,
                        int   *arrayOfLens);
```

## Syntax Elements

### **numOfTexts**

IN parameter

The number of EXPLAIN text strings.

### **arrayOfTexts**

IN parameter

An array containing the EXPLAIN text strings.

### **arrayOfLens**

IN parameter

An array containing the lengths of each EXPLAIN text string.

## Usage Notes

Hexplain has the following values for the type of EXPLAIN to be completed:

- 1 = simple
- 2 = verbose
- 3 = DBQL

This routine accepts multiple self-contained EXPLAIN text strings as input in order to handle a multi-row EXPLAIN plan from a foreign server. The routine provides the EXPLAIN plan to the parser which will display the multiple lines of the EXPLAIN plan.

This routine is callable on a PE vproc only by a table operator.

## FNC\_TblOpSetFormat

Sets attributes of the format of the input and output streams. This allows the contract function to specify the format of the data types to the parser.

### Syntax

```
void
FNC_TblOpSetFormat(char          *attribute,
                    int           streamno,
                    Stream_Direction_en direction,
                    void          *inValue,
                    int           inSize);
```

### Syntax Elements

#### *attribute*

IN parameter.

The format attribute to be set.

These are the valid attributes:

- "RECFMT"
- "TZTYPE"
- "CHARSETFMT"
- "REPUNSPTCHR"

"CHARSETFMT" and "REPUNSPTCHR" apply only to import table operators.

#### *streamno*

INparameter.

The stream number.

#### *direction*

IN parameter.

The stream direction: 'R' or 'W'.

***inValue***

IN parameter.

The location of the new value of the format attribute.

***inSize***

IN parameter.

The size in bytes of the new value pointed by *inValue*.

**Usage Notes**

- This routine is valid only when called within the contract function of a table operator.
- For "RECFMT" the default value is INDICFMT1, where the format is IndicData with row separator sentinels. When the format attribute is "RECFMT", the *inValue* buffer should have a value of type Stream\_Fmt\_en. All field-level formats impact the entire record.
- If data being imported from a foreign server contains characters unsupported by the database, you must use FNC\_ TblOpSetFormat and explicitly set "CHARSETFMT" and "REPUNSPTCHR" attributes.
- Format Attribute Values:

Format Attribute	Description
"RECFMT"	<p>Defines the record format. When the format attribute is "RECFMT", the <i>inValue</i> buffer should have a value of type Stream_Fmt_en. The Stream_Fmt_en enumeration is defined in the sqltypes_td.h header file with the following values:</p> <ul style="list-style-type: none"> <li>◦ INDICFMT1 = 1 IndicData with row separator sentinels.</li> <li>◦ INDICBUFFMT1 = 2 IndicData with NO row or partition separator sentinels.</li> </ul>
"TZTYPE"	<p>Used as an indicator to the database to receive from or send TIME/TIMESTAMP data to the table operator in a different format.</p> <ul style="list-style-type: none"> <li>◦ RAW = 0 as stored on the database file system</li> <li>◦ UTC = 1 as UTC</li> </ul>
"CHARSETFMT"	<ul style="list-style-type: none"> <li>◦ EVLDBC Signals that neither data conversion nor detection is needed.</li> <li>◦ EVLUTF16CHARSET Signals that the external data to be imported into the database are in UTF16 encoding.</li> <li>◦ EVLUTF8CHARSET Signals that the external data to be imported into the database are in UTF8 encoding.</li> </ul>
"REPUNSPTCHR"	<p>A boolean value that specifies what to do when an unsupported unicode character is detected in the external data to be imported into the database.</p> <ul style="list-style-type: none"> <li>◦ true Replaces the unsupported character with U+FFFD.</li> </ul>

Format Attribute	Description
	<ul style="list-style-type: none"> <li>◦ false</li> </ul> Return an error when an unsupported character is detected. This is the default behavior.

- You can map the database TIME and TIMESTAMP data types to the Hadoop STRING or the Oracle TIMESTAMP data type when importing or exporting data to these foreign servers.

The table operator can use FNC\_TbIOpSetFormat to set the tztype attribute as an indicator to the database to receive from or send TIMESTAMP data to the table operator in a native but adjusted format.

The tztype attribute is set as follows for the import and export operators:

- For Hadoop, the attribute is set to UTC.
- For Oracle, the attribute is set to UTC.

If the transform is off, the data will be transferred in Raw form which is the default for table operators and is consistent with standard UDFs.

tztype is a member of the structure FNC\_FmtConfig\_t defined in fnctypes.h as follows:

```
typedef struct
{
    int Stream_Fmt_en recordfmt; //enum - indicdata, fastload binary, delimited
    bool inlinelob; //inline or deferred
    bool UDTTransformsOff; //true or false
    bool PDTTransformsOff; //true or false
    bool ArrayTransformsOff; //true or false
    char auxinfo[128]; //For delimited text can contain the record
    separator, delimiter
    //specification and the field enclosure characters
    double inperc; //recommended percentage of buffer devoted to input rows
    bool inputnames; //send input column names to step
    bool outputnames; //send output column names to step
    TZType_en tztype; //enum - Raw or UTC
    int charsetfmt; // charset format of data to be imported into TD through QG
    bool replUnsprtedUniChar; /* true - replace unsupported unicode character
    encountered with U+FFFD when data is imported
    into TD through QG
    false - error out when unsupported unicode
    char encountered */
} FNC_FmtConfig_t;
```

TZType\_en is defined as follows:

```
typedef enum
{
```

```

    Raw = 0, /* as stored on TD File system */
    UTC = 1, /* as UTC */
} TZType_en;

```

For export, `FNC_TblOpSetInputColTypes` is called during the contract phase in the resolver and will use the `tztype` attribute to add the desired cast to the input `TIME` or `TIMESTAMP` column types.

The database converts the `TIME` and `TIMESTAMP` data to the session local time before casting to the character type, so when a `TIME` or `TIMESTAMP` column is being mapped to `charfix/charvar` as when mapping to the Hadoop `STRING` type, the data will transmit in session local time zone and no explicit casts are needed.

For import, when getting the input buffer from the table operator, `TIME` or `TIMESTAMP` data have to be converted to `Raw` form. There is no conversion needed for the import of Hadoop Strings to the database `TIME` or `TIMESTAMP` data types since it follows the normal conversion path from character to `TIME/TIMESTAMP` in the database.

---

#### Note:

Teradata does not recommend importing or exporting `TIME/TIMESTAMP` data for a database system with `timedatewzcontrol` flag `57 = 0`. For such systems, the `TIME/TIMESTAMP` data is stored in OS local time. The System/Session time zone is not set and the database does not apply any conversions on `TIME/TIMESTAMP` data when reading or writing from disk. Therefore, exporting such data reliably in the format desired by the foreign server is a problem and Teradata recommends that the Teradata-to-Hadoop connector feature not be used for such systems.

---

## FNC\_TblOpSetHashByDef

Allows the contract function writer to set the `HASH BY` specification when developing table operators.

### Syntax

```

void
FNC_TblOpSetHashByDef(int          streamno,
                      FNC_Names_t *colNames);

```

### Syntax Elements

#### *streamno*

the input stream number.

#### *colNames*

a pointer to the `HASH BY` metadata.

For details about the `FNC_Names_t` structure, see [Table Operator Data Structures](#).

## Usage Notes

This function can only run if called from the contract function. It will error if the stream number is invalid or the HASH BY metadata was already set.

## FNC\_TblOpSetInputColTypes

Sets casting statements on the input columns so that the data types are cast as indicated by the caller.

### Syntax

```
void
FNC_TblOpSetInputColTypes(int          streamNo,
                          FNC_TblOpColumnDef_t *colDefs)
```

### Syntax Elements

#### *streamNo*

IN parameter  
The input stream number.

#### *colDefs*

IN parameter  
A list of column definitions.

## Usage Notes

This routine is callable on a PE vproc only by a table operator.

## FNC\_TblOpSetLocalOrderByDef

Allows the contract function writer to set the ordering specification when developing table operators.

### Syntax

```
void
FNC_TblOpSetLocalOrderByDef(int          streamno,
                             FNC_Names_Ord_t *colNames);
```



## Syntax Elements

***streamno***

the input stream number.

***colNames***

a pointer to the LOCAL ORDER BY metadata.

For details about the FNC\_Names\_Ord\_t structure, see [Table Operator Data Structures](#).

## Usage Notes

This function can only run if called from the contract function. It will error if the stream number is invalid or the LOCAL ORDER BY metadata was already set.

# FNC\_TblOpSetOutputColDef

Communicates the output columns of an output stream to the parser.

## Syntax

```
void
FNC_TblOpSetOutputColDef(int          streamno,
                          FNC_TblOpColumnDef_t *coldefs);
```

## Syntax Elements

***streamno***

the output stream number.

***coldefs***

the columns definition.

For details about the FNC\_TblOpColumnDef\_t structure, see [Table Operator Data Structures](#).

## Usage Notes

If no contract function is specified, the parser sets the output columns as indicated by the RETURNS clause.

This function will error in the following cases:

- If both the contract function and an explicit column definition list are specified.
- The stream number is invalid.
- *coldefs* include invalid data types, invalid character set, or coldef->length is invalid for the number of columns in coldef->num\_columns.

**Example Using FNC\_TblOpSetOutputColDef**

See [C Table Operator](#) for an example of how to use this function.

**FNC\_TblOpWrite**

Allows table operator writers to write rows to spool. The function sets the write context to the next output row.

Returns an integer that indicates TBLOP\_SUCCESS, TBLOP\_NOROW, TBLOP\_ABORT, or TBLOP\_ERROR.

**Syntax**

```
int
FNC_TblOpWrite(FNC_TblOpHandle_t *handle);
```

**Syntax Elements*****handle***

a handle associated with an output stream. This should be the handle returned by FNC\_TblOpOpen.

For details about the FNC\_TblOpHandle\_t structure, see [Table Operator Data Structures](#).

**Usage Notes**

You must call FNC\_TblOpOpen first, then pass the handle returned from FNC\_TblOpOpen as an argument to FNC\_TblOpWrite.

If the stream was opened with the TBLOP\_NOOPTIONS option, the row fields can be directly set with function FNC\_TblOpBindAttributeByNdx, or directly set assigning:

```
handle->row->indicators, columnptr[index] and lengths[index].
```

If you call FNC\_TblOpOpen with the *options* field set to TBLOP\_RAWMODE, the raw row can be directly set through the handle structure:

```
Access handle->row to set the current raw row (length, IndicData format,
record body)
```

**Example Using FNC\_TblOpWrite**

See [C Table Operator](#) for an example of how to use this function.

## Related Information

For information about the `current_row_t` structure which stores information about the current row, see [Table Operator Data Structures](#).

## FNC\_TblOpWriteBuf

Provides access to the current output buffer.

Returns an integer that indicates `TBLOP_SUCCESS`, `TBLOP_ABORT`, or `TBLOP_ERROR`.

## Syntax

```
int
FNC_TblOpWriteBuf(BYTE *bufptr,
                  int    length)
```

## Syntax Elements

***bufptr***

a pointer to the current output buffer.

***length***

the length in bytes of the output buffer.

## Usage Notes

This function passes a pointer to the current output buffer. It is used for high performance direct buffer write access and requires that the buffer be in a standard row format. The buffer format will be `IndicData`.

## FNC\_TblOptOut

Called by a copy of a table function that does not want to participate in the process of returning rows.

IF the call is ...	THEN FNC_TblOptOut returns ...
successful	0.
not successful	-1. This can happen when a table function calls FNC_TblOptOut in the wrong mode or phase.

Use this library function when the return value of `FNC_GetPhase` is `TBL_MODE_CONST`, indicating that the `SELECT` statement invoked the table function with constant expression input arguments.

For example:

```
SELECT *
FROM TABLE (table_function_1('STRING_CONSTANT'))
AS table_1;
```

## Syntax

```
int
FNC_TblOptOut(void)
```

## Usage Notes

If every copy of the table function calls FNC\_TblOptOut, then the step ends as if no rows are created and the result is an empty derived table.

This function can only be called from within a table function. Calling this function from a scalar function, aggregate function, UDM, or external stored procedure results in an exception on the transaction.

Calling this function is valid only when the table function calls FNC\_GetPhase and gets a:

- return value of TBL\_MODE\_CONST for the mode
- value of TBL\_PRE\_INIT or TBL\_INIT for the processing phase

A table function can call FNC\_TblOptOut only once.

## Example Using FNC\_TblOptOut

```
FNC_Phase    Phase;

if (FNC_GetPhase(&Phase) == TBL_MODE_CONST)
{
    switch(Phase)
    {
        case TBL_PRE_INIT:

            switch (FNC_Tbl_FirstParticipant() )
            {
                case 1: /* participant */
                    return;
                case 0: /* not participant */
                    if (FNC_TblOptOut())
                        strcpy(sqlstate, "U0006"); /* an error return */
                    return;
                default: /* -1 or other error */
                    strcpy(sqlstate, "U0007");
                    return;
            }
        ...
    }
}
```

```

    }
    ...
}

```

## FNC\_Trace\_String

Returns the current setting of the UDF, UDM, or external stored procedure trace string set up with the SET SESSION FUNCTION TRACE statement.

### Syntax

```

void
FNC_Trace_String ( void *TraceStr )

```

### Syntax Elements

#### TraceStr

a pointer to memory that can hold the trace string.

The trace string is in the default character set of the user at the time the function was created or replaced.

The string is zero terminated. For UNICODE, the terminator is 16 bits of zero.

If the trace option is off, the string is set to NULL.

### Example Using FNC\_Trace\_String

```

SQL_TEXT TraceStr[258];

...

FNC_Trace_String(TraceStr);

```

### Related Information

For more information on SET SESSION FUNCTION TRACE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## FNC\_Trace\_Write

Provides backward compatibility for UDFs that were developed using older database releases to write trace output into a temporary trace table.

To write trace output into a temporary trace table from an external routine, use [FNC\\_Trace\\_Write\\_DL](#) instead.

## Syntax

```
void
FNC_Trace_Write ( int    argc,
                  void  *argv[] )
```

## Syntax Elements

### *argc*

the count of output arguments in the *argv* array.

This value should match the number of columns in the trace table beyond the first two mandatory columns that are used by the Teradata function trace subsystem.

### *argv*

an array of pointers to data to write to the columns in a temporary trace table.

Each array element in *argv* corresponds to a column in the trace table (beyond the first two mandatory columns). The order of the array elements matches the order of the columns in the trace table.

Consider the following trace table definition:

```
CREATE GLOBAL TEMPORARY TRACE TABLE Debug_Trace
  (AMP_vproc_ID BYTE(2)
  ,Sequence INTEGER
  ,Sum_X INTEGER
  ,Sum_Y INTEGER
  ,Trace_Text CHAR(30))
ON COMMIT DELETE ROWS;
```

The Teradata function trace subsystem writes values to the first two columns in the trace table. The first column identifies the AMP on which the function that called FNC\_Trace\_Write is running. The second column indicates the order in which the particular FNC\_Trace\_Write call was made on the AMP.

In the preceding example, the *argv* array elements match the columns in the trace table as follows:

Array Element	Column
<i>argv</i> [0]	Debug_Trace.Sum_X
<i>argv</i> [1]	Debug_Trace.Sum_Y
<i>argv</i> [2]	Debug_Trace.Trace_Text

The following rules apply:

IF ...	THEN ...
an <i>argv</i> element is null or <i>argv</i> has fewer elements than the corresponding columns in the trace table	the columns in the trace table corresponding to <i>argv</i> elements that are null or not supplied are set according to the following rules: If the corresponding column is defined as... <ul style="list-style-type: none"> <li>• nullable, then the value is set to NULL.</li> <li>• NOT NULL, then if the type of the column is... <ul style="list-style-type: none"> <li>◦ numeric, then the value is set to zero.</li> <li>◦ variable-length character, then the value is set to a zero-length string.</li> <li>◦ fixed-length character, then the value is set to all blanks.</li> <li>◦ fixed-length byte, then the value is set to all binary zeros.</li> </ul> </li> </ul>
<i>argv</i> has more elements than the corresponding columns in the trace table	the additional elements are ignored.

## Usage Notes

Trace output values are not validated. For example, stored values are incorrect when numeric values are too big for the receiving column or when character strings contain characters that are not valid.

If you store incorrect values, you might not be able to select data from the trace table. For example, if you write an invalid date to a `TIMESTAMP` column, a subsequent select on that data returns an invalid date error and no result. To read the bad `TIMESTAMP` data from the trace table and determine what is wrong, take the following steps:

1. Write another UDF that defines its input argument as a `TIMESTAMP` and converts it to a string result.
2. Use that UDF in the `SELECT` statement to read the `TIMESTAMP` data from the trace table to see what is wrong with it.

To avoid storing incorrect values that you might not be able to select, create the trace table with only character columns (beyond the first two mandatory columns) and format the output data using `sprintf`. For an example, see [Example: Debugging a UDF Using a Trace Table](#).

If an output string is too long for a corresponding character column, it is truncated without generating an error.

If no `SET SESSION FUNCTION TRACE` statement has been submitted to enable a trace table for trace output, an `FNC_Trace_Write` call is ignored.

`FNC_Trace_Write` can only be called from a UDF that always runs on an AMP.

## Example Using `FNC_Trace_Write`

```
INTEGER Sum_x;
INTEGER Sum_y;
void      *argv[3];
```

```

...

Sum_x = *In_data_x;
Sum_y = *In_data_y;

argv[0] = &Sum_x;
argv[1] = &Sum_y;
argv[2] = "The input values";

FNC_Trace_Write(3, argv);

```

## Related Information

For more information on CREATE GLOBAL TEMPORARY TRACE TABLE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## FNC\_Trace\_Write\_DL

Called by an external stored procedure, UDM, or UDF to write trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement.

## Syntax

```

void
FNC_Trace_Write_DL ( int      argc,
                    void    *argv[],
                    int      length[] )

```

## Syntax Elements

### *argc*

the count of output arguments in the *argv* array.

This value should match the number of columns in the trace table beyond the first two mandatory columns that are used by the Teradata function trace subsystem.

### *argv*

an array of pointers to data to write to the columns in a temporary trace table.

Each array element in *argv* corresponds to a column in the trace table (beyond the first two mandatory columns). The order of the array elements matches the order of the columns in the trace table.

Consider the following trace table definition:



```
CREATE GLOBAL TEMPORARY TRACE TABLE Debug_Trace
  (vproc_ID BYTE(2)
  ,Sequence INTEGER
  ,Sum_X INTEGER
  ,Sum_Y INTEGER
  ,Trace_Text CHAR(30))
ON COMMIT DELETE ROWS;
```

The Teradata function trace subsystem writes values to the first two columns in the trace table:

Column	Contents
1	PE or AMP vproc number on which the UDF, UDM, or external stored procedure is running
2	<p>If FNC_Trace_Write_DL is called from...</p> <ul style="list-style-type: none"> <li>An external stored procedure, UDM, or UDF running on a PE, then the value in the second column is a sequence number that increments by one for any call to FNC_Trace_Write_DL until the session logs off, regardless of which UDF, UDM, or external stored procedure makes the call.</li> </ul> <p>If the system restarts, the sequence number resets to one.</p> <ul style="list-style-type: none"> <li>a UDF running on an AMP, then the value in the second column is one more than the last sequential number written to the AMP for the trace table.</li> </ul>

In the preceding example, the *argv* array elements match the trace table columns as follows.

### ***length***

an array of the length, in bytes, of each output argument in the *argv* array.

The sum of the values of the *length* elements must be positive and must not exceed the size of a row in the trace table.

Array Element	Column
argv[0]	Debug_Trace.Sum_X
argv[1]	Debug_Trace.Sum_Y
argv[2]	Debug_Trace.Trace_Text

The following rules apply:

IF ...	THEN ...
<i>argv</i> has more elements than the corresponding columns in the trace table	the additional elements are ignored.
an <i>argv</i> element is null or	the columns in the trace table corresponding to <i>argv</i> elements that are null or not supplied are set according to the following rules:

IF ...	THEN ...
<i>argv</i> has fewer elements than the corresponding columns in the trace table	<p>If the corresponding column is defined as...</p> <ul style="list-style-type: none"> <li>• nullable, then the value is set to NULL.</li> <li>• NOT NULL, then if the type of the column is... <ul style="list-style-type: none"> <li>◦ numeric, then the value is set to zero.</li> <li>◦ variable-length character, then the value is set to a zero-length string.</li> <li>◦ fixed-length character, then the value is set to all blanks.</li> <li>◦ fixed-length byte, then the value is set to all binary zeros.</li> </ul> </li> </ul> <p>For <i>argv</i> elements that are null, corresponding elements in the <i>length</i> array must be zero.</p>

## Usage Notes

Trace output values are not validated. For example, stored values are incorrect when numeric values are too big for the receiving column or when character strings contain characters that are not valid.

If you store incorrect values, you might not be able to select data from the trace table. For example, if you write an invalid date to a `TIMESTAMP` column, a subsequent select on that data returns an invalid date error and no result. To read the bad `TIMESTAMP` data from the trace table and determine what is wrong, take the following steps:

1. Write another UDF that defines its input argument as a `TIMESTAMP` and converts it to a string result.
2. Use that UDF in the `SELECT` statement to read the `TIMESTAMP` data from the trace table to see what is wrong with it.

To avoid storing incorrect values that you might not be able to select, create the trace table with only character columns (beyond the first two mandatory columns) and format the output data using `sprintf`. For an example, see [Example: Debugging a UDF Using a Trace Table](#).

If an output string is too long for a corresponding character column, it is truncated without generating an error.

A UDF, UDM, or external stored procedure can call `FNC_Trace_Write_DL` as often as required.

The `FNC_Trace_Write_DL` call is ignored if any of the following conditions are true:

- No `SET SESSION FUNCTION TRACE` statement has been submitted to enable a trace table for trace output.
- The sum of the values of the *length* array elements is negative or exceeds the size of a row in the trace table.

The Teradata function trace subsystem writes values to the first two columns in a trace table. If you try to use the same trace table to simultaneously trace a UDF that runs on an AMP and an external stored procedure, UDM, or UDF that runs on a PE, the sequence number written to the second column will not be sequential. This is because the AMP bases the next sequence number on the sequence of the last row of the trace table and adds one to it no matter where the row originated. The rows inserted into the trace table from the PE are hashed to one AMP based on the PE vproc number and the current sequence number. Therefore if the

current sequence number for the PE is 5 and the trace row is added to AMP 1, then a trace write into that table from AMP 1 will have a sequence of 6. The best practice is to avoid simultaneously tracing on an AMP and PE.

### Example Using FNC\_Trace\_Write\_DL

```

INTEGER      Sum_x;
INTEGER      Sum_y;
CHARACTER_LATIN message[45];
void         *argv[3];
int          length[3];

...

Sum_x = *In_data_x;
Sum_y = *In_data_y;

argv[0] = &Sum_x;
length[0] = sizeof(INTEGER);

argv[1] = &Sum_y;
length[1] = sizeof(INTEGER);

message = strcpy((char *)message, "The input values");
argv[2] = &message;
length[2] = strlen((const char *)message);

FNC_Trace_Write_DL(3, argv, length);

```

### Related Information

For more information on CREATE GLOBAL TEMPORARY TRACE TABLE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## FNC\_UdtDeserialize

Deserializes data in the UDT's serialized format back into the UDT.

Returns 0 if the operation was successful, 1 otherwise.

### Syntax

```

int
FNC_UdtDeserialize (UDT_HANDLE      udt,

```

```

        BYTE
        FNC_LobLength_t    *buffer,
                           data_length)

```

## Syntax Elements

*udt*

the handle of the UDT.

*buffer*

the user buffer that contains the UDT value in its serialized format.

*data\_length*

the length of the data in the buffer.

## Usage Notes

When invoking FNC\_UdtDeserialize, you must pass in a buffer that contains all of the serialized data. FNC\_UdtDeserialize deserializes this data back into the UDT, but this must be done in a single call to FNC\_UdtDeserialize.

The query will fail if the UDT does not support serialize/deserialize.

# FNC\_UdtGetSerializeSize

Returns the actual size of the UDT in its serialized form.

## Syntax

```

FNC_LobLength_t
FNC_UdtGetSerializeSize(UDT_HANDLE    udt)

```

## Syntax Elements

*udt*

the handle of the UDT.

## Usage Notes

Use the size returned by this function to allocate the buffer that will hold the serialized or deserialized data in calls to FNC\_UdtSerialize and FNC\_UdtDeserialize.

The upper limit on the amount of memory that a UDF can allocate using FNC\_malloc is specified by the MallocLimit field in the cufconfig utility. For more information about cufconfig, see *Teradata Vantage™ - Database Utilities*, B035-1102.

The query will fail if the UDT does not support serialize/deserialize.

## FNC\_UdtSerialize

Returns the UDT in its serialized format and 0 if the operation was successful, -1 otherwise.

### Syntax

```
int
FNC_UdtSerialize (UDT_HANDLE      udt,
                  BYTE            *buffer,
                  FNC_LobLength_t num_bytes,
                  FNC_LobLength_t *actual_length)
```

### Syntax Elements

***udt***

the handle of the UDT.

***buffer***

the user-allocated buffer that will contain the serialized data.

***num\_bytes***

the number of bytes to read.

***actual\_length***

the actual number of bytes that were written to the buffer.

### Usage Notes

Before calling this function, do the following:

1. Call FNC\_UdtGetSerializeSize to get the size of the UDT in its serialized form.
2. Use the size returned by FNC\_UdtGetSerializeSize to allocate a buffer that will hold the serialized data. The size of the buffer must be large enough to hold all of the serialized data, which must be read in a single call to FNC\_UdtSerialize.
3. Pass this buffer to FNC\_UdtSerialize.

The query will fail if the UDT does not support serialize/deserialize.

## FNC\_UdtSerializeSupported

Returns 1 if the UDT supports serialization and deserialization, 0 otherwise.

## Syntax

```
int
FNC_UdtSerializeSupported(UDT_HANDLE udt)
```

## Syntax Elements

*udt*

the handle of the UDT.

## Usage Notes

The following UDTs support the serialize/deserialize interface. The formats that each UDT will use to serialize and deserialize data are as follows:

UDT	Format
ST_Geometry	The serialized format is opaque. It is designed to be compact and efficient to serialize and deserialize.
XML	The serialized format is opaque. It is designed to be compact and efficient to serialize and deserialize.
JSON	The serialized format is the text string that represents the JSON data, and it is equivalent to the transformed value (FromSQL/ToSQL) of the JSON data type.

## FNC\_Where\_Am\_I

Provides information about the specific node ID, vproc ID, vproc type, and task ID for the currently running external routine. This is useful if an external routine organizes GLOP data according to where it is invoked from.

Returns a pointer to a Vproc\_Info\_t structure.

## Syntax

```
Vproc_Info_t *
FNC_Where_Am_I (void);
```

## Vproc\_Info\_t

```
typedef struct
{
    unsigned short NumAMPs;
    unsigned short NumPEs;
    unsigned short NodeId;
```

```

    unsigned short VprocId;
    vproc_type     Vtype;
} Vproc_Info_t;

```

## Syntax Elements

### ***NumAMPs***

Number of online AMPs in the system.

### ***NumPEs***

Number of PEs in the system.

### ***NodeId***

Unique number of the node.

### ***VprocId***

Unique number of the vproc.

### ***Vtype***

Type of vproc, where the vproc\_type enumeration is defined as:

```

typedef enum
{
    UNK_VPROC = 0    /* vproc type unknown */
    AMP_VPROC = 1,   /* AMP vproc */
    PE_VPROC = 2     /* PE vproc */
} vproc_type;

```

## Usage Notes

FNC\_Where\_Am\_I provides information on the number vprocs of each type (AMP and PE), and what node, vproc, and type of vproc the external routine is running on. An external routine can use FNC\_Where\_Am\_I to help configure GLOP data that might require a different set up when executing on a PE instead of an AMP or data that might need to be divided up based on the number of AMP vprocs for a global copy operation.

### **Example Using FNC\_Where\_Am\_I**

Here is a code excerpt that shows how an external routine might use FNC\_Where\_Am\_I to find out what processor type it is executing on and only process transaction-related GLOP changes if it is running on an AMP.

```

#define TRAN_MAP  0
Sysinfo_t      SysInfoPtr;
GLOP_Map_t     *MyGLOP;
Vproc_Info_t   *MyVproc_info;
int            glop_stat;

    . . .

    MyVproc_info = FNC_Where_Am_I();

/* only bother with GLOP map if we are running on an AMP */
if (MyVproc->Vtype == AMP_VPROC)
{
    glop_stat = FNC_Get_GLOP_Map(&MyGLOP);

    if (glop_stat)
        ... process error: Not a member of any GLOP set
    if ( MyGLOP->GLOP[TRAN_MAP].GLOP_Ptr == NULL)
    {
        ... process error: Transaction GLOP does not exist
    };
    SysInfoPtr = MyGLOP->GLOP[TRAN_MAP].GLOP_ptr;

    . . .

}
else
{
    /* do non AMP processing here */
}

```



# Java Application Classes

The following sections discuss Java application classes that Teradata provides in the javFnc.jar archive for use by Java UDFs and external stored procedures (collectively called *external routines*).

Only public methods that are currently supported appear in this section.

Java external routines can use the following application classes that Teradata provides in the javFnc.jar archive.

IF the external routine is a ...	AND it ...	THEN use ...
Java UDF or external stored procedure	uses an SQL BLOB type parameter and specifies a data access clause of NO SQL in the CREATE /REPLACE FUNCTION or CREATE/REPLACE PROCEDURE statement	com.teradata.fnc.Blob
Java UDF or external stored procedure	uses an SQL CLOB type parameter and specifies a data access clause of NO SQL in the CREATE /REPLACE FUNCTION or CREATE/REPLACE PROCEDURE statement	com.teradata.fnc.Clob
Java UDF or external stored procedure	needs to obtain session information related to the current execution of the UDF or external stored procedure	com.teradata.fnc.DbsInfo
Java UDF or external stored procedure	writes trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement for debugging purposes	com.teradata.fnc.TraceObj
Java UDF or external stored procedure	needs to access query band information for a session, transaction, and/or profile	com.teradata.fnc.QueryBand
Java UDF	is an aggregate UDF	com.teradata.fnc.Phase
		com.teradata.fnc.Context
Java UDF	is a table UDF	com.teradata.fnc.Tbl
		com.teradata.fnc.AmplInfo
		com.teradata.fnc.NodeInfo
Java UDF	is a table operator	com.teradata.fnc.operator
		com.teradata.fnc.operator.Metadata
		com.teradata.fnc.runtime

Java Application Class	Description
<code>com.teradata.fnc.Array</code>	ARRAY/VARRAY type parameters to a Java routine are mapped to <code>java.sql.Array</code> . The <code>java.sql.Array</code> interface provides methods for bringing the data of an SQL ARRAY value to the client as either an array or a <code>ResultSet</code> object. The <code>java.sql.Array</code> interface is implemented by the <code>com.teradata.fnc.Array</code> class.
<code>com.teradata.fnc.SQLXML</code>	XML type parameters to a Java routine are mapped to <code>java.sql.SQLXML</code> . The <code>java.sql.SQLXML</code> interface provides methods for bringing the data of an SQL XML value to the client as a <code>String</code> , a <code>Reader</code> or <code>Writer</code> , or as a <code>Stream</code> . The <code>java.sql.SQLXML</code> interface is implemented by the <code>com.teradata.fnc.SQLXML</code> class.
<code>com.teradata.fnc.ST_Geometry</code>	ST_Geometry data types can be optionally mapped to the <code>com.teradata.fnc.ST_Geometry</code> Java class. Geospatial data can be retrieved in either the Well-Known Text (WKT) or Well-Known Binary (WKB) format as a CLOB (for WKT) or a BLOB (for WKB) from an input ST_Geometry value, and either format can be written to a return ST_Geometry via the same respective CLOB and BLOB types.
<code>com.teradata.fnc.Struct</code>	Structured UDT and Period data types are mapped to the <code>java.sql.Struct</code> interface. The <code>java.sql.Struct</code> interface is implemented by the <code>com.teradata.fnc.Struct</code> class.

## Class Path

To use the classes that Teradata provides for UDFs and external stored procedures, the search path for Java classes must include the directory containing the `javFnc.jar` archive. The default location for the archive is in the `bin` directory of the Teradata software distribution:

```
/usr/tdbms/bin
```

## com.teradata.fnc.AMPInfo

Defines methods that return AMP-specific information that a local copy of a table function or table operator can use to configure itself to use the correct resources.

### Syntax

```
public class com.teradata.fnc.AMPInfo extends java.lang.Object{
    public com.teradata.fnc.AMPInfo()
        throws java.io.IOException, java.sql.SQLException;
    public int getNodeId();
    public int getAMPId();
    public boolean lowestAMPOnNode();
}
```

## AMPInfo Constructor

Constructs an instance of AMPInfo. The AMPInfo constructor is valid only in a table operator or in a table UDF, during any mode or phase.

### getNodeId()

Returns the unique number of the current node.

### getAMPId()

Returns the unique number of the current AMP.

### lowestAMPOnNode()

Returns true if the current AMP is the lowest AMP on the same node as the invoking AMP, otherwise returns false.

If this method is invoked from a table operator that is associated with a map, then lowestAMPOnNode() returns true if the current AMP is the lowest AMP on the same node within the specified map, otherwise it returns false.

This information is useful for a table UDF or table operator that allows only the lowest AMP vproc to participate.

## com.teradata.fnc.Array

ARRAY/VARRAY type parameters to a Java routine are mapped to java.sql.Array. The java.sql.Array interface provides methods for bringing the data of an SQL ARRAY value to the client as either an array or a ResultSet object. The java.sql.Array interface is implemented by the com.teradata.fnc.Array class.

The following sections describe the methods implemented by this class for operating on ARRAYS.

### free()

Frees the Array object and releases the resources that it holds.

---

#### Note:

Both 1-D and N-D arrays are supported for this method.

---

### Syntax

```
void free()
    throws SQLException
```

## Exceptions

Throws SQLException if an error occurs while attempting to free the array and release its resources. A database specific code "9743 (ERRUDFJAVARRAY) <Failed to free Array object>" is returned.

## getArray()

Retrieves the contents of the SQL ARRAY value designated by this Array object in the form of an array in the Java programming language.

---

### Note:

- Teradata only supports standard mapping.
  - Both 1-D and N-D arrays are supported for this method.
- 

An array in the Java programming language that contains the ordered elements of the SQL ARRAY value designated by this Array object. For N-D arrays, the elements are returned in row-major order.

## Syntax

```
Object getArray()
           throws SQLException
```

## Exceptions

Throws SQLException if an error occurs while attempting to access the array. A database specific code "9743 (ERRUDFJAVARRAY)<Failed to retrieve elements of the array>" is returned.

## getArray(Map)

Retrieves the contents of the SQL ARRAY value designated by this Array object in the form of an array in the Java programming language.

---

### Note:

- Teradata only supports standard mapping. Any custom mapping provided will be ignored.
  - Both 1-D and N-D arrays are supported for this method.
- 

An array in the Java programming language that contains the ordered elements of the SQL ARRAY value designated by this Array object. For N-D arrays, the elements are returned in row-major order.

## Syntax

```
Object getArray(Map<String,Class<?>> map)
           throws SQLException
```

## Syntax Elements

### *map*

A `java.util.Map` object that contains mappings of SQL type names to classes in the Java programming language.

## Exceptions

Throws `SQLException` if an error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY)<Failed to retrieve elements of the array>” is returned.

## getArray(long, int)

Retrieves a slice of the SQL ARRAY value designated by this Array object, beginning with the specified *index* and containing up to *count* successive elements of the SQL array.

### Note:

- Teradata only supports standard mapping.
- This method is supported for 1-D arrays only.

An array containing up to *count* consecutive elements of the SQL ARRAY value, beginning with element *index*.

Note that this routine may return less than *count* number of elements in the array. Any uninitialized elements in the array are not returned.

## Syntax

```
Object getArray(long index,
                 int count)
           throws SQLException
```

## Syntax Elements

### *index*

The array index of the first element to retrieve. The first element is at index 1.

***count***

The number of successive SQL array elements to retrieve.

**Exceptions**

Throws SQLException in the following cases:

- The method is called for an N-D array. A database specific code “9743(ERRUDFJAVARRAY)<Method> cannot be called for N-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY)<Failed to retrieve elements of the array>” is returned.

**getArray(long, int, Map)**

Retrieves a slice of the SQL ARRAY value designated by this Array object, beginning with the specified *index* and containing up to *count* successive elements of the SQL array.

**Note:**

- Teradata only supports standard mapping. Any custom map provided in the call will be ignored.
- This method is supported for 1-D arrays only.

An array containing up to *count* consecutive elements of the SQL ARRAY value, beginning with element *index*.

Note that this routine may return less than *count* number of elements in the array. Any uninitialized elements in the array are not returned.

**Syntax**

```
Object getArray(long  index,
                int    count,
                Map<String,Class<?>>  map)
                throws SQLException
```

**Syntax Elements*****index***

The array index of the first element to retrieve. The first element is at index 1.

***count***

The number of successive SQL array elements to retrieve.

**map**

A `java.util.Map` object that contains SQL type names and the classes in the Java programming language to which they are mapped.

**Exceptions**

Throws `SQLException` in the following cases:

- The method is called for an N-D array. A database specific code “9743 (ERRUDFJAVARRAY) <Method> cannot be called for N-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to retrieve elements of the array>” is returned.

**getBaseType()**

Retrieves the JDBC type of the elements in the array designated by this Array object.

A constant from the class `Types` that is the type code for the elements in the array designated by this Array object .

**Syntax**

```
int getBaseType()
    throws SQLException
```

**Exceptions**

Throws `SQLException` if an error occurs while attempting to access the base type. A database specific code “9743 (ERRUDFJAVARRAY) <Array element type could not be retrieved>” is returned.

**getBaseTypeName()**

Retrieves the SQL type name of the elements in the array designated by this Array object. If the elements are a built-in type, it returns the database-specific type name of the elements. If the elements are a user-defined type (UDT), this method returns the fully qualified SQL type name.

A `String` that is the database-specific name for a built-in base type or the fully qualified SQL type name for a base type that is a UDT.

**Syntax**

```
String getBaseTypeName()
    throws SQLException
```

## Exceptions

Throws SQLException if an error occurs while attempting to access the type name. A database specific code “9743 (ERRUDFJAVARRAY) <Array type name could not be retrieved>” is returned.

## getNumDimensions()

Retrieves the number of dimensions defined for a given Array. Possible values correspond to the number of dimensions supported for Teradata ARRAY types. This method supports both 1-D and N-D arrays.

An int value containing the number of dimensions defined for the given Array type.

## Syntax

```
int getNumDimensions()
    throws SQLException
```

## Exceptions

Throws SQLException if an error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to retrieve number of Array dimensions>” is returned.

## getResultSet()

Retrieves a result set that contains the elements of the SQL ARRAY value designated by this Array object.

### Note:

- Teradata only supports standard mapping.
- This method only supports 1-D arrays.

A ResultSet object containing one row for each of the elements in the array designated by this Array object.

The result set contains one row for each array element, with two columns in each row:

- The first column is of type Integer and stores the index into the array for that element, with the first array element being at index 1.
- The second column stores the element value.

The rows are in ascending order corresponding to the order of the indices.

## Syntax

```
ResultSet getResultSet()
    throws SQLException
```



## Exceptions

Throws SQLException in the following cases:

- The method is called for an N-D array. A database specific code “9743(ERRUDFJAVARRAY)<Method> cannot be called for N-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743(ERRUDFJAVARRAY)<Failed to retrieve result set of Array elements>” is returned.

## getResultSet(Map)

Retrieves a result set that contains the elements of the SQL ARRAY value designated by this Array object.

### Note:

- Teradata only supports standard mapping. Any custom map provided in the call will be ignored.
- This method only supports 1-D arrays.

A ResultSet object containing one row for each of the elements in the array designated by this Array object.

The result set contains one row for each array element, with two columns in each row:

- The first column is of type Integer and stores the index into the array for that element, with the first array element being at index 1.
- The second column stores the element value.

The rows are in ascending order corresponding to the order of the indices.

## Syntax

```
ResultSet getResultSet(Map<String,Class<?>> map)
                throws SQLException
```

## Syntax Elements

### *map*

The mapping of SQL user-defined types to classes in the Java programming language.

## Exceptions

Throws SQLException in the following cases:

- The method is called for an N-D array. A database specific code “9743(ERRUDFJAVARRAY)<Method> cannot be called for N-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743(ERRUDFJAVARRAY)<Failed to retrieve result set of Array elements>” is returned.

## getResultSet(long,int)

Retrieves a result set holding the elements of the subarray that starts at *index* and contains up to *count* successive elements.

---

### Note:

- Teradata only supports standard mapping.
  - This method only supports 1-D arrays.
- 

A `ResultSet` object containing up to *count* consecutive elements of the SQL array designated by this Array object, starting at *index*.

The result set has one row for each element of the SQL array with the first row containing the element at *index*. The result set has up to *count* rows in ascending order based on the indices. Each row has two columns:

- The first column stores the index into the array for that element.
- The second column stores the element value.

Note that this routine may return less than *count* number of elements in the array. Any uninitialized elements in the array are not returned.

### Syntax

```
ResultSet getResultSet(long  index,
                        int    count)
                        throws SQLException
```

### Syntax Elements

#### *index*

The array index of the first element to retrieve. The first element is at index 1.

#### *count*

The number of successive SQL array elements to retrieve.

### Exceptions

Throws `SQLException` in the following cases:

- The method is called for an N-D array. A database specific code "9743(ERRUDFJAVARRAY)<Method> cannot be called for N-D Arrays >" is returned.
- An error occurs while attempting to access the array. A database specific code "9743(ERRUDFJAVARRAY)<Failed to retrieve result set of Array elements>" is returned.

## getResultSet(long, int, Map)

Retrieves a result set holding the elements of the subarray that starts at *index* and contains up to *count* successive elements.

---

### Note:

- Teradata only supports standard mapping. Any custom map provided in the call will be ignored.
  - This method only supports 1-D arrays.
- 

A `ResultSet` object containing up to *count* consecutive elements of the SQL array designated by this Array object, starting at *index*.

The result set has one row for each element of the SQL array with the first row containing the element at *index*. The result set has up to *count* rows in ascending order based on the indices. Each row has two columns:

- The first column stores the index into the array for that element.
- The second column stores the element value.

Note that this routine may return less than *count* number of elements in the array. Any uninitialized elements in the array are not returned.

### Syntax

```
ResultSet getResultSet(long  index,
                        int    count,
                        Map<String,Class<?>>  map)
                        throws SQLException
```

### Syntax Elements

#### *index*

The array index of the first element to retrieve. The first element is at index 1.

#### *count*

The number of successive SQL array elements to retrieve.

#### *map*

The Map object that contains the mapping of SQL type names to classes in the Java programming language.

### Exceptions

Throws `SQLException` in the following cases:

- The method is called for an N-D array. A database specific code “9743(ERRUDFJAVARRAY)< <Method> cannot be called for N-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY)<Failed to retrieve result set of Array elements>” is returned.

## getResultSet\_nD()

Retrieves a result set holding all the elements of the n-D array.

---

### Note:

- Teradata only supports standard mapping.
  - This method supports only N-D arrays.
  - This method is a Teradata extension.
- 

A `ResultSet` object containing all the elements of the N-D array.

The result set has one row for each element of the SQL array, with the first row containing the first present element in the array, in row-major order.

The result set has up to the number of rows corresponding to the number of elements present in the array. For n-D arrays, the column type of the first row is of `List<Integer>`, where the list contains  $n$  coordinates corresponding to the particular element being referenced.

For example, for a 2-D array, the `List<Integer>` column for a given row would consist of 2 elements  $(n,m)$ . For a 5-D array, the `List<Integer>` column for a given row would consist of 5 elements  $(a,b,c,d,e)$ .

The index values for the first column of the `ResultSet` are based on the lower and upper bounds defined for the array. Therefore, if the array type is defined with lower bounds that are some other value than 1, the index values for the first column of the `ResultSet` will begin with that same lower bound accordingly.

The rows containing the array elements are output in row-major order.

Any uninitialized elements in the array are not returned.

### Syntax

```
ResultSet getResultSet_nD() throws SQLException
```

### Exceptions

Throws `SQLException` in the following cases:

- The method is called for a 1-D array. A database specific code “9743 (ERRUDFJAVARRAY)< <Method> cannot be called for 1-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to retrieve result set of Array elements>” is returned.

## getResultSet\_nD(List<Integer>, List<Integer>)

Retrieves a result set holding the elements of the slice of the n-D array that starts at the index specified by the coordinates in List `lowerBounds` and contains up to the total number of elements in the slice delimited by the coordinates in List `upperBounds`.

---

### Note:

- Teradata only supports standard mapping.
  - This method supports N-D arrays only.
  - This method is a Teradata extension.
- 

A `ResultSet` object containing up to the full slice specified of the SQL array designated by this Array object, starting at the element referred to by `lowerBound`.

The result set has one row for each element of the SQL array, with the first row containing the element at the lower bound specified by `lowerBounds`.

The result set has up to the number of rows corresponding to the number of elements contained in the slice between `lowerBounds` and `upperBounds`. For n-D arrays, the column type of the first row is of `List<Integer>`, where the list contains *n* coordinates corresponding to the particular element being referenced.

For example, for a 2-D array, the `List<Integer>` column for a given row would consist of 2 elements (*n,m*). For a 5-D array, the `List<Integer>` column for a given row would consist of 5 elements (*a,b,c,d,e*).

The index values for the first column of the `ResultSet` are based on the lower and upper bounds defined for the array. Therefore, if the array type is defined with lower bounds that are some other value than 1, then the index values for the first column of the `ResultSet` will begin with that same lower bound accordingly.

The rows containing the array elements are output in row-major order.

Note that this routine may return less than the full slice of elements in the array. Any uninitialized elements in the array are not returned.

### Syntax

```
ResultSet getResultSet_nD(List<Integer>  lowerBounds,
                           List<Integer>  upperBounds)
                           throws SQLException
```

### Syntax Elements

#### *lowerBounds*

The array index of the first element to retrieve from the slice.

***upperBounds***

The array index of the last element to retrieve from the slice.

**Exceptions**

Throws SQLException in the following cases:

- The method is called for a 1-D array. A database specific code “9743 (ERRUDFJAVARRAY) <Method> cannot be called for 1-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to retrieve result set of Array elements>” is returned.

**getUDFOutputArray()**

Creates a new instance of com.teradata.fnc.Array.

**Note:**

- This method is supported for both 1-D and N-D arrays. However, this method is provided mainly for support of N-D arrays because the com.teradata.fnc.Array class provides some additional method signatures especially for N-D arrays. For 1-D array, it is recommended to use any generic implementation of java.sql.Array instead, since 1-D array support conforms to the JDBC standard.
- This method is valid for scalar and aggregate UDFs only. For table UDFs and external stored procedures, you should create a new Array object via the routine’s INOUT or OUT parameters and update its attributes accordingly.
- This method is a Teradata extension.

A new object of type com.teradata.fnc.Array.

The new com.teradata.fnc.Array object is initialized with the type information provided for the scalar or aggregate UDF’s return value.

If the array is defined as DEFAULT NULL, the new com.teradata.fnc.Array object will have all its elements initialized to NULL. Otherwise, no elements are initialized.

**Syntax**

```
com.teradata.fnc.Array getUDFOutputArray()
                        throws SQLException
```

**Exceptions**

Throws SQLException with the following database specific code and message: 9743 (ERRUDFJAVARRAY) <Unsupported Array element type passed to getUDFOutputArray>.

## getXSPInoutArrayForNull(int)

Creates a new instance of `com.teradata.fnc.Array` when a NULL is passed in as the INOUT parameter of a Java external stored procedure.

---

### Note:

- This method is supported for both 1-D and N-D arrays. However, this method is provided mainly for support of N-D arrays because the `com.teradata.fnc.Array` class provides some additional method signatures especially for N-D arrays. For 1-D arrays, it is recommended to use any generic implementation of `java.sql.Array` instead, because 1-D array support conforms to the JDBC standard.
  - This method is a Teradata extension.
- 

A new object of type `com.teradata.fnc.Array`.

The new `com.teradata.fnc.Array` object is initialized with the type information provided for the INOUT parameter of the external stored procedure.

If the array is defined as DEFAULT NULL, the new `com.teradata.fnc.Array` object will have all its elements initialized to NULL. Otherwise, no elements are initialized.

### Syntax

```
com.teradata.fnc.Array getXSPInoutArrayForNull(int param_index)
                                throws Exception
```

### Syntax Elements

#### *param\_index*

The index of the particular INOUT parameter as it appears in the list of parameters for the Java external stored procedure, starting from 0.

## setArray(Object, long, int)

Sets a slice of the SQL ARRAY value designated by this Array object, using the element values passed in *arrayElements*, beginning with the specified *index* and setting up to *count* successive elements of the SQL array.

**Note:**

- This method supports 1-D arrays only.
- If *index* is -1 or *count* is -1, the Array object is completely overwritten starting from the first element. Otherwise, the routine sets elements in the Array object starting from *index*, up to *count* elements. It may be possible that less than *count* elements are set.

**Syntax**

```
void setArray(Object  arrayElements,
              long   index,
              int    count)
              throws SQLException
```

**Syntax Elements*****arrayElements***

An Object consisting of a Java array of element values to be set.

***index***

The array index of the first element to set. The first element is at index 1.

***count***

The number of successive SQL array elements to set.

**Exceptions**

Throws SQLException in the following cases:

- The method is called for an N-D array. A database specific code “9743 (ERRUDFJAVARRAY) <Method> cannot be called for N-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to set elements of the array>” is returned.

**setArray\_nD(Object, List<Integer>, List<Integer>)**

Sets a slice of the SQL ARRAY value designated by this Array object using the element values passed in *arrayElements*. The slice of elements to be modified is designated by lower bound coordinates specified in *lowerBounds* and upper bound coordinates specified in *upperBounds*.



**Note:**

- This method supports N-D arrays only.
- If the coordinates specified by *lowerBounds* and *upperBounds* are set to -1, the Array object is completely overwritten starting from the lowest coordinate value.
- This routine may set less than the full slice of elements in the array. Any section of the slice outside of the boundaries of the array are ignored.

**Syntax**

```
void setArray(Object  arrayElements,
              List<Integer>  lowerBounds,
              List<Integer>  upperBounds)
              throws SQLException
```

**Syntax Elements*****arrayElements***

An Object consisting of a Java array of element values to be set.

***lowerBounds***

The array index of the first element to set in the slice.

***upperBounds***

The array index of the last element to set in the slice.

**Exceptions**

Throws SQLException in the following cases:

- The method is called for a 1-D array. A database specific code “9743 (ERRUDFJAVARRAY)< <Method> cannot be called for 1-D Arrays >” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to set elements of the array>” is returned.

**setResultSet(ResultSet)**

Sets an SQL 1-D ARRAY value designated by this Array object with values provided in the input *ResultSet*. Element values in the Array object are set starting from the lowest value of the first column to the highest.

**Note:**

- This method supports 1-D arrays only.
- Fewer than the maximum number of elements specified in the *ResultSet* may be set in the Array, in case the Array was constructed with fewer elements.

**Syntax**

```
void setResultSet(ResultSet  arrayResultSet)
                throws SQLException
```

**Syntax Elements*****arrayResultSet***

A *ResultSet* consisting of two columns, an index and an element value.

The result set passed as input must contain one row for each array element, with two columns in each row:

- The first column is of type Integer and stores the index into the array for that element.
- The second column stores the element value.

The rows are in ascending order corresponding to the order of the indices.

**Exceptions**

Throws *SQLException* in the following cases:

- The method is called for an N-D array. A database specific code “9743 (ERRUDFJAVARRAY) <Method> cannot be called for N-D Arrays>” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to set Array elements with provided ResultSet>” is returned.

**setResultSet\_nD(ResultSet)**

Sets an SQL N-D ARRAY value designated by this Array object with values provided in the input *ResultSet*. Element values in the Array object are set starting from the lowest value of the first column (according to row-major order) to the highest.

**Note:**

- This method supports N-D arrays only.
- Fewer than the maximum number of elements specified in the *ResultSet* may be set in the Array, in case the Array was constructed with fewer elements.

## Syntax

```
void setResultSet_nD(ResultSet  arrayResultSet)
                        throws SQLException
```

## Syntax Elements

### *arrayResultSet*

A `ResultSet` consisting of two columns, an index and an element value.

The result set passed as input must contain one row for each array element, with two columns in each row:

- The first column is of type `List<Integer>`, where the list contains *n* coordinates corresponding to a particular element.
- The second column stores the element value.

The rows are in ascending order corresponding to the order of the indices.

## Exceptions

Throws `SQLException` in the following cases:

- The method is called for a 1-D array. A database specific code “9743 (ERRUDFJAVARRAY)< <Method> cannot be called for 1-D Arrays>” is returned.
- An error occurs while attempting to access the array. A database specific code “9743 (ERRUDFJAVARRAY) <Failed to set Array elements with provided ResultSet>” is returned.

## Examples: Using `java.sql.Array` Interface For ARRAY Parameter

### Example: Using `java.sql.Array` Interface For 1-D ARRAY Parameter

```
CREATE TYPE INTARRAY AS INTEGER ARRAY[5];

CREATE PROCEDURE GET_ARRAY(IN A1 INTARRAY)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.get_array';

public static void get_array(java.sql.Array A1) throws SQLException
{
    System.out.println ("Array is of type "+A1.getBaseTypeName());
    System.out.println ("Array element type:" + A1.getBaseType());
}
```

```

    // get Array elements
    int[] values = (Int[]) A1.getArray();
    for (int i=0; i<values.length; i++)
    {
        System.out.println("element " + i + " = "+values[i]);
    }
}

```

### Example: Using java.sql.Array Interface For *n*-D ARRAY Parameter

This example uses the java.sql.Array interface for an *n*-D ARRAY type parameter and gets elements as a result set.

```

CREATE TYPE SHOTS AS INTEGER ARRAY[1:3][1:3][1:3];

CREATE PROCEDURE GET_ND_ARRAY(IN A1 SHOTS)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.get_nd_array';

public static void get_nd_array(java.sql.Array A1) throws SQLException
{
    System.out.println ("Array is of type "+A1.getBaseTypeName());
    System.out.println ("Array element type:" + A1.getBaseType());
    System.out.println
        ("Array num dimensions:" + A1.getNumDimensions());

    // get Array elements as a result set
    ResultSet nDArrayElements;
    List<Integer> lowerBounds = new ArrayList<Integer>();
    List<Integer> upperBounds = new ArrayList<Integer>();
    lowerBounds.add(1);
    lowerBounds.add(1);
    lowerBounds.add(1);
    upperBounds.add(3);
    upperBounds.add(3);
    upperBounds.add(3);

    nDArrayElements = A1.getResultSet_nD(lowerBounds, upperBounds);

    while (nDArrayElements.next()) {
        List<Integer> elementNum = nDArrayElements.getObject(1);
        int elementItem = nDArrayElements.getInt(2);
    }
}

```

```

        System.out.println ("Array element index: " + elementNum +
                           "element value: " + elementItem);
    }
}

```

## com.teradata.fnc.Blob

Provides all the necessary information for the user to interact with a Blob object when it is a parameter in a Java external routine. The Blob object can be mapped from other objects such as JSON, XML, or ST\_Geometry objects.

### Syntax

```

public class com.teradata.fnc.Blob extends java.lang.Object
implements java.sql.Blob{
    public void free() throws java.sql.SQLException;
    public java.io.InputStream getBinaryStream()
        throws java.sql.SQLException;
    public java.io.InputStream getBinaryStream(long, long)
        throws java.sql.SQLException;
    public byte[] getBytes(long, int) throws java.sql.SQLException;
    public static java.sql.Blob getUDFOutputBlob()
        throws java.sql.SQLException;
    public static java.sql.Blob getXSPInoutBlobForNull(int)
        throws Exception;
    public long length() throws java.sql.SQLException;
    public long position(byte[], long) throws java.sql.SQLException;
    public long position(java.sql.Blob, long)
        throws java.sql.SQLException;
    public java.io.OutputStream setBinaryStream(long)
        throws java.sql.SQLException;
    public int setBytes(long, byte[]) throws java.sql.SQLException;
    public int setBytes(long, byte[], int, int)
        throws java.sql.SQLException;
    public void truncate(long) throws java.sql.SQLException;
}

```

### free()

Releases the resources held by the Blob object. The object is invalid once free() is called. After free() is called, any attempt to invoke a method other than free() results in an SQLException.

## getBinaryStream()

Retrieves the BLOB value designated by this Blob object as a stream.

## getBinaryStream(long *pos*, long *length*)

Retrieves all or part of the BLOB value designated by this Blob object as a stream.

Returns a `java.io.InputStream` object that contains up to *length* consecutive bytes from the BLOB value designated by this Blob object, starting with the byte at position *pos*.

### Syntax

```
getBinaryStream(long pos, long length)
```

### Syntax Elements

#### *pos*

The ordinal position of the first byte in the BLOB value to be extracted.

The first byte is at position 1.

#### *length*

The number of consecutive bytes to be copied.

## getBytes(long *pos*, int *length*)

Retrieves all or part of the BLOB value that this Blob object represents as an array of bytes.

The result byte array contains up to *length* consecutive bytes from the BLOB value designated by this Blob object, starting with the byte at position *pos*.

If  $pos + length - 1$  is larger than the length of the BLOB in bytes, the length of the resulting byte array is less than *length*.

### Syntax

```
getBytes(long pos, int length)
```

### Syntax Elements

#### *pos*

The ordinal position of the first byte in the BLOB value to be extracted.

The first byte is at position 1.

***length***

The number of consecutive bytes to be copied.

**Exceptions**

IF ...	THEN <code>getBytes()</code> throws an <code>SQLException</code> to ...
there is an error accessing the BLOB value	indicate an unexpected failure while reading LOB data, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7851</li> </ul>
the value of the <i>pos</i> argument is less than 1 -or- the value of the <i>length</i> argument is negative	indicate an invalid argument, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "22023"</li> <li>• <code>vendorCode</code> Field = 7859</li> </ul>
the value of the <i>length</i> argument is too large for the current JVM to allocate the required memory in this method	indicate that JVM heap memory has run out, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7856</li> </ul>

**getUDFOutputBlob()**

Retrieves the Blob object representing the return value of a Java UDF that returns a Blob. This method can only be called from a scalar or aggregate Java UDF. If called from an aggregate Java UDF, it may only be called in the `Phase.AGR_FINAL` execution phase.

A `java.sql.Blob` object representing the return Blob value of the Java UDF.

**Exceptions**

IF the caller is ...	THEN <code>getUDFOutputBlob()</code> throws an <code>SQLException</code> to ...
not a scalar or aggregate Java UDF	indicate that the caller is not a scalar or aggregate Java UDF, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7863</li> </ul>
a UDF that does not return a Blob or Clob type	indicate that the caller does not return a Blob or Clob, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7862</li> </ul>
a UDF that returns a Clob type	indicate that the caller actually returns a Clob, setting the <code>SQLException</code> fields as follows:

IF the caller is ...	THEN getUDFOutputBlob() throws an SQLException to ...
	<ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7866</li> </ul>
an aggregate Java UDF that returns a Blob, but the method is not called in the Phase. AGR_FINAL aggregation phase	indicate that the method was called from an incorrect aggregation phase, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7864</li> </ul>

## getXSPInoutBlobForNull(int *index*)

Returns a Blob object representing the INOUT parameter of a Java external stored procedure when a Null object is passed in for the parameter.

The `java.sql.Blob` object that represents the INOUT parameter of a Java external stored procedure when a Null object is passed in for the parameter. This object can be used to set the return value for the INOUT parameter that was passed in as a NULL.

### Syntax

```
java.sql.Blob getXSPInoutBlobForNull(int index)
                                throws Exception
```

### Syntax Elements

#### *index*

The index of the particular INOUT parameter as it appears in the list of parameters for the Java external stored procedure, starting from 0.

### Exceptions

Throws Exception if any error occurs during the process which could be either IOException or SQLException depending on the root cause of the error.

## length()

Returns the number of bytes in the BLOB value designated by this Blob object.



## position(byte[] *pattern*, long *start*)

Retrieves the byte position at which the specified byte array pattern begins within the BLOB value that this Blob object represents. The search for *pattern* begins at position *start*.

If the search is successful, position() returns the position at which the pattern appears; otherwise, position returns a value of -1.

### Syntax

```
position(byte[] pattern, long start)
```

### Syntax Elements

#### *pattern*

The byte array containing the pattern for which to search.

#### *start*

The position at which to begin searching.

The first position is 1.

### Exceptions

IF ...	THEN position() throws an SQLException to ...
there is an error accessing the BLOB value	indicate a failure while reading LOB data, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7851</li> </ul>
the value of the <i>pattern</i> argument is null -or- the value of the <i>start</i> argument is less than 1	indicate an invalid argument, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "22023"</li> <li>• vendorCode Field = 7859</li> </ul>
the JVM cannot allocate enough memory for the method to search for the specified pattern	indicate that available JVM heap memory has run out, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7856</li> </ul>

## position(java.sql.Blob *pattern*, long *start*)

Retrieves the byte position at which the Blob pattern begins within the BLOB value that this Blob object represents. The search for *pattern* begins at position *start*.

### Note:

This method is reserved for a future release.

### Syntax

```
position(java.sql.Blob pattern, long start)
```

### Syntax Elements

#### *pattern*

The byte array containing the pattern for which to search.

#### *start*

The position at which to begin searching.

The first position is 1.

### Exceptions

This method throws an SQLException to indicate that a Java external routine called a method that is not yet supported, setting the SQLException fields as follows.

SQLState Field	vendorCode Field
"0A000"	7858

## setBinaryStream(long *pos*)

Retrieves a stream that can be used to append to the end of the BLOB value that this Blob object represents.

setBinaryStream() returns a java.io.OutputStream object to which data can be written.

### Syntax

```
setBinaryStream(long pos)
```

## Syntax Elements

### *pos*

The *pos* argument is reserved for future use.

## Exceptions

If the value of the *pos* argument is less than 1, `setBinaryStream()` throws an `SQLException` to indicate that a Java external routine called a method with invalid parameters, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"22023"	7859

## setBytes(long *pos*, byte[] *bytes*)

Appends the given array of bytes to the end of the BLOB value that this Blob object represents and returns the number of bytes written.

If successful, `setBytes()` returns the number of bytes written.

## Syntax

```
setBytes(long pos, byte[] bytes)
```

## Syntax Elements

### *pos*

Reserved for future use.

### *bytes*

The array of bytes to append to the BLOB value that this Blob object represents.

## Exceptions

IF ...	THEN <code>setBytes()</code> throws an <code>SQLException</code> to ...
there is an error accessing the BLOB value	indicate a failure while writing data into a LOB, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "TS000"</li> <li>vendorCode Field = 7852</li> </ul>
the <i>bytes</i> argument is null -or-	indicate a call to a Blob or Clob method with invalid parameters, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "22023"</li> <li>vendorCode Field = 7859</li> </ul>

IF ...	THEN <code>setBytes()</code> throws an <code>SQLException</code> to ...
the value of the <i>pos</i> argument is less than 1	
the JVM cannot allocate enough memory for the method to write the bytes to the Blob	indicate that available JVM heap memory has run out, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7856</li> </ul>

## `setBytes(long pos, byte[] bytes, int offset, int len)`

Appends all or part of the given byte array to the end of the BLOB value that this Blob object represents and returns the number of bytes written.

If successful, `setBytes()` returns the number of bytes appended.

### Syntax

```
setBytes(long pos, byte[] bytes, int offset, int len)
```

### Syntax Elements

#### *pos*

Reserved for future use.

#### *bytes*

The array of bytes to append to the BLOB value that this Blob object represents.

#### *offset*

The offset into the *bytes* argument at which to start reading the bytes to append.

#### *len*

The number of bytes from the *bytes* argument to append to the BLOB value.

### Exceptions

IF ...	THEN <code>setBytes()</code> throws an <code>SQLException</code> to ...
there is an error accessing the BLOB value	indicate a failure while writing to the BLOB, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7852</li> </ul>

IF ...	THEN <code>setBytes()</code> throws an <code>SQLException</code> to ...
the JVM cannot allocate enough memory for the method to write the bytes to the Blob	indicate that the available JVM heap memory has run out, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7856</li> </ul>
one of the following: <ul style="list-style-type: none"> <li>• the <i>bytes</i> argument is null</li> <li>• the value of the <i>offset</i> argument is negative</li> <li>• the value of the <i>len</i> argument is negative</li> <li>• the value of the <i>pos</i> argument is less than 1</li> <li>• the value of <i>offset</i> + <i>len</i> is larger than the length of the <i>bytes</i> argument</li> </ul>	indicate a call to a Blob or Clob method with an invalid argument, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "22023"</li> <li>• <code>vendorCode</code> Field = 7859</li> </ul>

## truncate(long *len*)

Truncate the BLOB value that this Blob object represents to be *len* bytes in length.

### Note:

This method is reserved for a future release.

## Syntax

```
truncate(long len)
```

## Syntax Elements

### *len*

The number of bytes from the *bytes* argument to append to the BLOB value.

## Exceptions

This method throws an `SQLException` to indicate that a Java external routine called a method that is not yet supported, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"0A000"	7858

## com.teradata.fnc.Clob

Provides all the necessary information for the user to interact with a Clob object when it is a parameter in a Java external routine. The Clob object can be mapped from other objects such as JSON or ST\_Geometry objects.

### Syntax

```
public class com.teradata.fnc.Clob extends java.lang.Object
implements java.sql.Clob{
    public void free() throws java.sql.SQLException;
    public java.io.InputStream getAsciiStream()
        throws java.sql.SQLException;
    public java.io.Reader getCharacterStream()
        throws java.sql.SQLException;
    public java.io.Reader getCharacterStream(long, long)
        throws java.sql.SQLException;
    public java.lang.String getSubString(long, int)
        throws java.sql.SQLException;
    public static java.sql.Clob getUDFOutputClob()
        throws java.sql.SQLException;
    public static java.sql.Clob getXSPInoutClobForNull(int)
        throws Exception;
    public long length() throws java.sql.SQLException;
    public long position(java.lang.String, long)
        throws java.sql.SQLException;
    public long position(java.sql.Clob, long)
        throws java.sql.SQLException;
    public java.io.OutputStream setAsciiStream(long)
        throws java.sql.SQLException;
    public java.io.Writer setCharacterStream(long)
        throws java.sql.SQLException;
    public int setString(long, java.lang.String)
        throws java.sql.SQLException;
    public int setString(long, java.lang.String, int, int)
        throws java.sql.SQLException;
    public void truncate(long) throws java.sql.SQLException
}
```

### free()

Releases the resources held by the Clob object. The object is invalid once free() is called. After free() is called, any attempt to invoke a method other than free() results in an SQLException.

## getAsciiStream()

Retrieves the CLOB value designated by this Clob object as a standard ASCII stream.

## getCharacterStream()

Retrieves the CLOB value designated by this Clob object as a stream of characters.

## getCharacterStream(long *pos*, long *length*)

Retrieves all or part of the CLOB value designated by this Clob object as a stream of characters.

Returns a java.io.Reader object that contains up to *length* consecutive characters from the CLOB value designated by this Clob object, starting with the character at position *pos*.

### Syntax

```
getCharacterStream(long pos, long length)
```

### Syntax Elements

#### *pos*

The position of the first character in the CLOB value to be extracted.

The first character is at position 1.

#### *length*

The number of consecutive characters to be copied.

## getSubString(long *pos*, int *length*)

Retrieves all or part of the CLOB value that this Clob object represents as a string of characters.

The result string contains up to *length* consecutive characters from the CLOB value designated by this Clob object, starting with the character at position *pos*.

If the value of *pos* + *length* - 1 is larger than the length of the CLOB in characters, the length of the resulting string is less than *length*.

### Syntax

```
getSubString(long pos, int length)
```

## Syntax Elements

### *pos*

The position of the first character in the CLOB value to be extracted.

The first character is at position 1.

### *length*

The number of consecutive characters to be copied.

## Exceptions

IF ...	THEN <code>getSubString()</code> throws an <code>SQLException</code> to ...
there is an error accessing the CLOB value	indicate a failure while the reading from Clob data, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7851</li> </ul>
the value of the <i>pos</i> argument is less than 1 -or- the value of the <i>length</i> argument is negative	indicate an invalid argument, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "22023"</li> <li>• <code>vendorCode</code> Field = 7859</li> </ul>
the value of the <i>length</i> argument is too large for the current JVM to allocate the required memory for this function	indicate that available JVM heap memory has run out, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7856</li> </ul>

## getUDFOutputClob()

Retrieves the Clob object representing the return value of a Java UDF that returns a Clob. This method can only be called from a scalar or aggregate Java UDF. If called from an aggregate Java UDF, it may only be called in the `Phase.AGR_FINAL` execution phase. The character set of the Clob object returned by `getUDFOutputClob()` is the same as that of the Clob value that the caller returns.

A `java.sql.Clob` object representing the return Clob value of the Java UDF.



## Exceptions

IF the caller is ...	THEN getUDFOutputClob() throws an SQLException to ...
not a scalar or aggregate Java UDF	indicate that the caller is not a scalar or aggregate Java UDF, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7863</li> </ul>
a UDF that does not return a Blob or Clob type	indicate that the caller does not return a Blob or Clob, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7862</li> </ul>
a UDF that returns a Blob type	indicate that the caller actually returns a Blob, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7866</li> </ul>
an aggregate Java UDF that returns a Clob, but the method is not called in the Phase. AGR_FINAL aggregation phase	indicate that the method was called from an incorrect aggregation phase, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7864</li> </ul>

## getXSPInoutClobForNull(int *index*)

Returns a Clob object representing the INOUT parameter of a Java external stored procedure when a Null object is passed in for the parameter.

The `java.sql.Clob` object that represents the INOUT parameter of a Java external stored procedure when a Null object is passed in for the parameter. This object can be used to set the return value for the INOUT parameter that was passed in as a NULL.

## Syntax

```
java.sql.Clob getXSPInoutClobForNull(int index)
                                throws Exception
```

## Syntax Elements

### *index*

The index of the particular INOUT parameter as it appears in the list of parameters for the Java external stored procedure, starting from 0.

## Exceptions

Throws Exception if any error occurs during the process which could be either IOException or SQLException depending on the root cause of the error.

## length()

Returns the number of characters in the CLOB value designated by this Clob object.

## position(java.lang.String *searchstr*, long *start*)

Retrieves the character position at which the specified search string begins within the CLOB value that this Clob object represents. The search for *searchstr* begins at position *start*.

If the search is successful, position() returns the position at which the search string begins; otherwise, position() returns a value of -1.

## Syntax

```
position(java.lang.String searchstr, long start)
```

## Syntax Elements

### *searchstr*

The string for which to search.

The first character is at position 1.

### *start*

The position at which to begin searching.

The first position is 1.

## Exceptions

IF ...	THEN position() throws an SQLException to ...
there is an error accessing the CLOB value	indicate a failure while reading LOB data, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "TS000"</li> <li>vendorCode Field = 7851</li> </ul>
the <i>searchstr</i> argument is null -or- the value of the <i>start</i> argument is less than 1	indicate an invalid argument, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "22023"</li> <li>vendorCode Field = 7859</li> </ul>

IF ...	THEN position() throws an SQLException to ...
the JVM cannot allocate enough memory for the method to search for the specified string	indicate that available JVM heap memory has run out, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>• SQLState Field = "TS000"</li> <li>• vendorCode Field = 7856</li> </ul>

## position(java.sql.Clob searchClob, long start)

Retrieves the character position at which the Clob argument *searchClob* begins within the CLOB value that this Clob object represents. The search for *searchClob* begins at position *start*.

### Note:

This method is reserved for a future release.

## Exceptions

This method throws an SQLException to indicate that a Java external routine called a method that is not yet supported, setting the SQLException fields as follows.

SQLState Field	vendorCode Field
"0A000"	7858

## setAsciiStream(long pos)

Retrieves a stream that can be used to append standard ASCII characters to the end of the CLOB value that this Clob object represents.

setAsciiStream() returns a java.io.OutputStream object to which standard ASCII can be written.

## Syntax

```
setAsciiStream(long pos)
```

## Syntax Elements

### pos

The *pos* argument is reserved for future use.

## Exceptions

If the value of the *pos* argument is less than 1, `setAsciiStream()` throws an `SQLException` to indicate that a Java external routine called a Blob or Clob method with invalid arguments, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"22023"	7859

## setCharacterStream(long pos)

Retrieves a stream that can be used to append a stream of Unicode characters to the end of the CLOB value that this Clob object represents.

`setCharacterStream()` returns a stream to which Unicode characters can be written.

## Syntax

```
setCharacterStream(long pos)
```

## Syntax Elements

*pos*

The *pos* argument is reserved for future use.

## Exceptions

If the value of the *pos* argument is less than 1, `setCharacterStream()` throws an `SQLException` to indicate that a Java external routine called a Blob or Clob method with invalid arguments, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"22023"	7859

## setString(long pos, java.lang.String str)

Appends the given string to the end of the CLOB value that this Clob object represents and returns the number of characters written.

If successful, `setString()` returns the number of characters written.

## Syntax

```
setString(long pos, java.lang.String str)
```

## Syntax Elements

### *pos*

The *pos* argument is reserved for future use.

### *str*

The string to append to the CLOB value that this Clob object represents.

## Exceptions

IF ...	THEN setString() throws an SQLException to ...
there is an error accessing the CLOB value	indicate a failure while writing data into a LOB, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "TS000"</li> <li>vendorCode Field = 7852</li> </ul>
the <i>str</i> argument is null -or- the value of the <i>pos</i> argument is less than 1	indicate that a Java external routine called a Blob or Clob method with invalid arguments, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "22023"</li> <li>vendorCode Field = 7859</li> </ul>
the JVM cannot allocate enough memory for the method to write the string to the Clob	indicate that the available JVM heap memory has run out, setting the SQLException fields as follows: <ul style="list-style-type: none"> <li>SQLState Field = "TS000"</li> <li>vendorCode Field = 7856</li> </ul>

## setString(long pos, java.lang.String str, int offset, int len)

Appends all or part of the given string to the end of the CLOB value that this Clob object represents and returns the number of characters written.

If successful, setString() returns the number of characters appended.

## Syntax

```
setString(long pos, java.lang.String str, int offset, int len)
```

## Syntax Elements

### *pos*

Reserved for future use.

### *str*

The string to append to the CLOB value that this Clob object represents.

### *offset*

The offset into the *str* argument at which to start reading the characters to append.

### *len*

The number of characters from the *str* argument to append to the CLOB value.

## Exceptions

IF ...	THEN <code>setString()</code> throws an <code>SQLException</code> to ...
there is an error accessing the Clob value	indicate a failure while writing data into a LOB, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7852</li> </ul>
the JVM cannot allocate enough memory for the method to write the string to the Clob	indicate that available JVM heap memory has run out, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "TS000"</li> <li>• <code>vendorCode</code> Field = 7856</li> </ul>
one of the following: <ul style="list-style-type: none"> <li>• the <i>str</i> argument is null</li> <li>• the value of the <i>offset</i> argument is negative</li> <li>• the value of the <i>len</i> argument is negative</li> <li>• the value of the <i>pos</i> argument is less than 1</li> <li>• the value of <i>offset</i> + <i>len</i> is larger than the length of the <i>str</i> argument</li> </ul>	indicate that a Java external routine called a Blob or Clob method with an invalid argument, setting the <code>SQLException</code> fields as follows: <ul style="list-style-type: none"> <li>• <code>SQLState</code> Field = "22023"</li> <li>• <code>vendorCode</code> Field = 7859</li> </ul>

## `truncate(long len)`

Truncate the CLOB value that this Clob object represents to be *len* characters in length.

**Note:**

This method is reserved for a future release.

**Syntax**

```
truncate(long len)
```

**Syntax Elements*****len***

The number of characters from the *str* argument to append to the CLOB value.

**Exceptions**

This method throws an SQLException to indicate that a Java external routine called a method that is not yet supported, setting the SQLException fields as follows.

SQLState Field	vendorCode Field
"0A000"	7858

**com.teradata.fnc.Context**

Provides a way for an aggregate UDF to access intermediate storage to combine data passed in during the various aggregation phases.

A Java aggregate UDF must define the second parameter as a Context type.

**Syntax**

```
public class com.teradata.fnc.Context extends java.lang.Object{
    public static final int ERRUDFJAGRINITCTXOVFLW;
    public static final int ERRUDFJAGRINTOUTRNG;
    public static final int ERRUDFJAGRSETOBJOVFLW;
    public static final int ERRUDFJAGRSETBYTESOVFLW;
    public long getVersion();
    public int getFlags();
    public java.lang.Object getObject(int)
        throws java.lang.ClassNotFoundException, java.sql.SQLException;
    public void setObject(int, java.lang.Object)
        throws java.io.IOException, java.sql.SQLException;
    public void initCtx(java.lang.Object)
        throws java.io.IOException, java.sql.SQLException;
```

```

public long  getGroupCount();
public long  getWindowSize();
public long  getPreWindow();
public long  getPostWindow();
public void  initCtx(int)
    throws java.io.IOException, java.sql.SQLException;
public byte[] getBytes(int)
    throws java.io.IOException, java.sql.SQLException;
public void  setBytes(int, byte[])
    throws java.io.IOException, java.sql.SQLException;
}

```

## getVersion()

Reserved for future use.

## getFlags()

Reserved for future use.

## getObject(int *objNum*)

Returns data that was stored in one of two intermediate aggregate storage areas.

If the specified storage area contains no data, getObject() returns a null object.

### Syntax

```
getObject(int objNum)
```

### Syntax Elements

#### *objNum*

The *objNum* input argument specifies which aggregation storage area to retrieve intermediate results from.

Value	Meaning
1	Retrieve intermediate results from the first aggregate storage area. The data that the UDF retrieves was stored from a call to Context.setObject() during a previous aggregation phase. The UDF uses the previously stored data during the Phase.AGR_DETAIL, Phase.AGR_COMBINE and Phase.AGR_FINAL aggregation phases.
2	Retrieve intermediate results from the second aggregate storage area.



Value	Meaning
	The data that the UDF retrieves is stored by Vantage. The UDF uses the data during the Phase.AGR_COMBINE aggregation phase.

## Exceptions

If the value of *objNum* is out of range, getObject() throws an SQLException to indicate that the input parameter of Context.getObject(int) is out of range, setting the SQLException fields as follows.

SQLState Field	vendorCode Field
"TS000"	7844

## setObject(int objNum, java.lang.Object obj)

Stores intermediate results specified by *obj* into the aggregate storage area.

Each time the UDF is called during the AGR\_DETAIL aggregation phase, it must accumulate the row data passed in through arguments into the intermediate aggregate storage area for the specific group. Each group being aggregated to has a separate intermediate storage area.

## Syntax

```
setObject(int objNum, java.lang.Object obj)
```

## Syntax Elements

### *objNum*

Aggregate UDFs must pass a value of 1 for the *objNum* argument. Other values are reserved for future releases.

### *obj*

Intermediate results to store into the aggregate storage area.

## Exceptions

If the serialized size of *obj* is larger than the size specified in the CLASS AGGREGATE clause of the CREATE FUNCTION or REPLACE FUNCTION statement for the UDF, setObject() throws an SQLException to indicate an overflow condition, setting the SQLException fields as follows.

SQLState Field	vendorCode Field
"TS000"	7845

## initCtx(java.lang.Object *obj*)

Allocates and initializes the aggregate intermediate storage area according to the serialized size of the *obj* input argument.

An aggregate UDF uses this method in the Phase.AGR\_INIT phase of aggregation.

### Syntax

```
initCtx(java.lang.Object obj)
```

### Syntax Elements

#### *obj*

Object whose serialized size the function uses to allocate and initialize the aggregate intermediate storage area.

If the input parameter *obj* has object type fields, the UDF must initialize those fields to non-null values before calling initCtx(). If *obj* has array type fields, the UDF must initialize the array itself and each array element to non-null values before calling initCtx().

### Exceptions

If the serialized size of *obj* is larger than the size specified in the CLASS AGGREGATE clause of the CREATE FUNCTION or REPLACE FUNCTION statement for the UDF, initCtx() throws an SQLException to indicate that aggregate storage allocation failed because requested size is too big. The SQLException fields are set to the following values.

SQLState Field	vendorCode Field
"TS000"	7843

## getGroupCount()

Reserved for future use.

## getWindowSize()

Reserved for future use.

## getPreWindow()

Reserved for future use.

## getPostWindow()

Reserved for future use.

## initCtx(int *length*)

Allocates and initializes the aggregate intermediate storage area according to the length specified by the *length* input argument.

An aggregate UDF uses this method in the Phase.AGR\_INIT phase of aggregation.

### Syntax

```
initCtx(int length)
```

### Syntax Elements

#### *length*

Length the function uses to allocate and initialize the aggregate intermediate storage area.

### Exceptions

If *length* is larger than the size specified in the CLASS AGGREGATE clause of the CREATE FUNCTION or REPLACE FUNCTION statement for the UDF, initCtx() throws an SQLException to indicate that aggregate storage allocation failed because requested size is too big. The SQLException fields are set to the following values.

SQLState Field	vendorCode Field
"TS000"	7843

## getBytes(int *objNum*)

Returns data that was stored in one of two intermediate aggregate storage areas.

If the specified storage area contains no data, getBytes() returns a null object.

This method is similar to Context.getObject() but provides a faster interface.

### Syntax

```
getBytes(int objNum)
```

## Syntax Elements

### *objNum*

Specifies which aggregation storage area to retrieve intermediate results from, as follows:

Value	Meaning
1	Retrieve intermediate results from the first aggregate storage area. The data that the UDF retrieves was stored from a call to <code>Context.setByte()</code> during a previous aggregation phase. The UDF uses the previously stored data during the <code>Phase.AGR_DETAIL</code> , <code>Phase.AGR_COMBINE</code> and <code>Phase.AGR_FINAL</code> aggregation phases.
2	Retrieve intermediate results from the second aggregate storage area. The data that the UDF retrieves is stored by Vantage. The UDF uses the data during the <code>Phase.AGR_COMBINE</code> aggregation phase.

## Exceptions

If the value of *objNum* is out of range, `getBytes()` throws an `SQLException` to indicate that the input parameter of `Context.getObject(int)` is out of range, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"TS000"	7844

## `setBytes(int objNum, byte[] data)`

Stores intermediate results specified by *data* into the aggregate storage area.

Each time the UDF is called during the `AGR_DETAIL` aggregation phase, it must accumulate the row data passed in through arguments into the intermediate aggregate storage area for the specific group. Each group being aggregated to has a separate intermediate storage area.

## Syntax

```
setBytes(int objNum, byte[] data)
```

## Syntax Elements

### *objNum*

Aggregate UDFs must pass a value of 1 for the *objNum* argument. Other values are reserved for future releases.

***data***

If *data* specifies a null byte array, `setBytes()` sets the aggregate storage area to zero.

**Exceptions**

If the serialized size of *data* is larger than the size specified in the CLASS AGGREGATE clause of the CREATE FUNCTION or REPLACE FUNCTION statement for the UDF, `setBytes()` throws an `SQLException` to indicate an overflow condition, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"TS000"	7847

**com.teradata.fnc.DbsInfo**

Provides a way to obtain information such as user account, user name, user ID and session number of the UDF or external stored procedure.

**Syntax**

```
public class com.teradata.fnc.DbsInfo extends java.lang.Object{
    public com.teradata.fnc.DbsInfo();
    public static java.lang.String getUserAccount();
    public static java.lang.String getUserAccount_EON();
    public static java.lang.String getUsername();
    public static java.lang.String getUsername_EON();
    public static long getUserId();
    public static short getStatementNo();
    public static short getHost();
    public static long getSessionNo();
    public static long getRequestNo();
    public static java.lang.String getTraceString();
    public static void traceWrite(java.lang.Object[]);
}
```

**getUserAccount() [Deprecated]**

This method is deprecated because it truncates object names to 30 characters. However, it remains available to support legacy applications. For current and future development, use the corresponding method that includes "EON" in its name. For example, use `getUsername_EON()` instead of `getUsername()`.

Returns a maximum of 30 characters specifying the current user account. Use instead `getUserAccount_EON()` for a longer name. The string is encoded according to the server character set that the UDF or external stored procedure was defined to expect and produce.

For more information about object naming, see *Teradata Vantage™ - SQL Fundamentals*, B035-1141.

## **getUserAccount\_EON()**

Returns a maximum of 128 characters which is at most 513 bytes of data specifying the current user account in UTF-8 encoding. Note that `getUserAccount_EON()` returns a null-terminated character string.

## **getUserName() [Deprecated]**

This method is deprecated because it truncates object names to 30 characters. However, it remains available to support legacy applications. For current and future development, use the corresponding method that includes "EON" in its name. For example, use `getUserName_EON()` instead of `getUserName()`.

Returns a maximum of 30 characters specifying the current user name. Use instead `getUserName_EON()` for a longer name. The string is encoded according to the server character set that the UDF or external stored procedure was defined to expect and produce.

## **getUserName\_EON()**

Returns a maximum of 128 characters which is at most 513 bytes of data specifying the current user name in UTF-8 encoding. Note that `getUserName_EON()` returns a null-terminated character string.

## **getUserId()**

Returns the current user ID value.

## **getStatementNo()**

Returns the current statement number for the request.

## **getHost()**

Returns the host ID.

## **getSessionNo()**

Returns the current session number.

## getRequestNo()

Returns the current client request number.

## getTraceString()

Returns the current setting of the UDF or external stored procedure trace string set up with the SET SESSION FUNCTION TRACE statement. If the trace option is off, the string is set to NULL.

## traceWrite(java.lang.Object[] argv)

Called by a UDF or external stored procedure to write trace output into a temporary trace table defined by a CREATE GLOBAL TEMPORARY TRACE TABLE statement.

Each object in the *argv* array corresponds to a column in the trace table (beyond the first two mandatory columns). The order of the objects in the *argv* array matches the order of the columns in the trace table.

To use DbsInfo.traceWrite to write trace output into a temporary trace table, follow these steps:

1. For each column in the trace table, create a Java object that maps to the SQL data type of the column and set the value of the object to the value you want to write to the column.

For details on how Java objects map to the SQL data types of a temporary trace table column, see [TraceObj Constructor](#).

2. Wrap each Java object using the TraceObj wrapper class.

For details on TraceObj, see [com.teradata.fnc.TraceObj](#).

3. Put the TraceObj objects into an Object array and pass the array to DbsInfo.traceWrite.

DbsInfo.traceWrite converts the value of each object in the array to its corresponding SQL data type value and writes the value to the trace table.

FOR more information on ...	SEE ...
debugging a Java UDF or external stored procedure using trace tables	<a href="#">Debugging Using Trace Tables</a> .
CREATE GLOBAL TEMPORARY TRACE TABLE	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

## com.teradata.fnc.NodeInfo

Defines methods that return node IDs and AMP IDs for all online AMP vprocs, allowing table functions and table operators to configure themselves to run on specific AMPs.

## Syntax

```
public class com.teradata.fnc.NodeInfo extends java.lang.Object{
    public com.teradata.fnc.NodeInfo()
        throws java.io.IOException, java.sql.SQLException;
    public int getNumAMPNodes();
    public int getNumAMPs();
    public int[] getNodeIds();
    public int[] getAMPIds();
}
```

## NodeInfo Constructor

Constructs an instance of NodeInfo. The NodeInfo constructor is valid only for Java table UDFs and table operators.

## getNumAMPNodes()

Returns the total number of AMPs on the same node as the invoking AMP.

If this method is invoked from a table operator that is associated with a map, then getNumAMPNodes() returns the total number of AMPs on the same node within the specified map.

## getNumAMPs()

Returns the total number of AMPs that are in online or hold state within the same node as the invoking AMP.

If this method is invoked from a table operator that is associated with a map, then getNumAMPs() returns the total number of AMPs that are in online or hold state within the specified map and on the same node as the invoking AMP.

## getNodeIds()

Returns an array of node information, where each array element is the unique number of a node.

The number of elements in the array can be determined by NodeInfo.getNumAMPs().

The array elements map one-to-one with the array elements from NodeInfo.getAMPIds(). For example, the second array element from NodeInfo.getNodeIds() is the node number for the AMP in the second array element from NodeInfo.getAMPIds().

Node numbers occur in ascending order and each node number is repeated for the total number of AMPs on that particular node.



**Note:**

In the event of a node failure, a HSN (hot standby node) takes over the online AMP vprocs from the failed node. In this case, `getNodeIds()` returns the node ID of the HSN instead of the failed node. Therefore, you may get results that are not in the order expected with regards to the node ID.

**getAMPIds()**

Returns an array of AMP information, where each array element is the unique number of an AMP.

The number of elements in the array can be determined by `NodeInfo.getNumAMPs()`.

AMP numbers occur in ascending order for each node. For example, if the first node has four AMP vprocs, the first four array elements provide the AMP numbers, in ascending order, for the first node. Similarly, the first four array elements from `NodeInfo.getNodeIds()` provide the node number of the first node.

**com.teradata.fnc.Operator**

Obtains blocks of rows for input and export.

**Syntax**

```
public interface RawResultSet
```

**getResultSet****Syntax**

```
java.sql.ResultSet getResultSet()
```

**read**

Obtains blocks of rows for input.

**Syntax**

```
int read(byte[] readbuf)
    throws java.sql.SQLException
```

**write**

Obtains blocks of rows for export.

**Syntax**

```
void write(byte[] writebuf,
           int datalen)
    throws java.sql.SQLException
```

**com.teradata.fnc.operator.Metadata**

The com.teradata.fnc.operator.Metadata class provides a means to describe metadata information about all of the columns in an input stream to a table operator.

The com.teradata.fnc.operator.Metadata class implements the methods of the java.sql.ResultSetMetaData interface. It also provides the additional four methods.

**getJSONFormat(int)**

Returns the storage format of the JSON column.

**Syntax**

```
byte getJSONFormat(int column)
    throws SQL Exception
```

**getPeriodType(int)**

Returns the Period value for the specified column. The following are the valid values for the Period type value:

```
NOT_PERIOD = 0
PERIOD_DATE = 1
PERIOD_TIME = 2
PERIOD_TIME_WTZ = 3
PERIOD_TIMESTAMP = 4
PERIOD_TIMESTAMP_WTZ = 5
```

The method throws SQLException if an error occurs while attempting to access the Period type information.

**Syntax**

```
int getPeriodType(int column)
    throws SQL Exception
```

## getTeradataColumnType(int)

Returns the data type of the column.

### Syntax

```
int getTeradataColumnType(int column)
    throws SQL Exception
```

## getUdtTypeName(int)

Returns a String value that is the UDT name of the column if it is a UDT column.

The method throws SQLException if an error occurs while attempting to access the UDT name information.

### Syntax

```
String getUdtTypeName(int column)
    throws SQL Exception
```

## com.teradata.fnc.Phase

Provides a way for Java aggregate UDFs to determine the current phase of execution and what action to perform.

A Java aggregate UDF must define the first parameter as a Phase type.

### Syntax

```
public class com.teradata.fnc.Phase extends java.lang.Object{
    public static final int   AGR_INIT;
    public static final int   AGR_DETAIL;
    public static final int   AGR_COMBINE;
    public static final int   AGR_FINAL;
    public static final int   AGR_NODATA;
    public static final int   AGR_MOVINGTRAIL;
    public int   getPhase();
}
```

## getPhase()

Returns the current aggregation phase. The aggregation phase determines how to process the data passed in.

The return value is one of the Phase constants.

Value	Meaning
Phase.AGR_INIT	This is the first time Vantage invokes the aggregate UDF for an aggregation group. For a window aggregate UDF, this phase is triggered once per partition at the start of a new aggregation group or first row. The UDF must: <ul style="list-style-type: none"> <li>• Allocate intermediate storage and initialize it.</li> <li>• Process the first detail passed into the UDF.</li> </ul>
Phase.AGR_DETAIL	This phase is triggered every time the forward row progresses. In this phase, Vantage calls the UDF once for each row to be aggregated for each group. The UDF must combine the input data with the intermediate storage defined for the group.
Phase.AGR_COMBINE	This phase combines results from different AMPs for a specific group. The UDF must combine the data for the two intermediate storage areas passed in. This phase is not applicable for window aggregate UDFs.
Phase.AGR_MOVINGTRAIL	This phase is only applicable for window aggregate UDFs for the moving window type (noncumulative, nonreporting window type). This phase is triggered by the last few rows of a moving window when the forward pointer to the window reaches the end of the group or end of the file. The phase does not provide any row or value to the UDF, but it is mainly used to indicate to the UDF that we are reaching the end of the group or file. The UDF can use this phase to adjust the necessary internal count or related values to reflect the actual size as the window diminishes towards the end of the group or file.
Phase.AGR_FINAL	No more input is expected for the group. The UDF must produce the final result for the group.
Phase.AGR_NODATA	This phase is only presented when there is absolutely no data to aggregate.

## com.teradata.fnc.QueryBand

Provides a way to retrieve query band name-value pairs that have been set on a session, transaction, or profile to identify the originating source of queries and help manage task priorities and track system use.

### Syntax

```
public class com.teradata.fnc.QueryBand extends java.lang.Object{
    public com.teradata.fnc.QueryBand();
    public java.lang.String getQueryBand();
    public java.lang.String getQueryBand(int);
    public int getQueryBandLen(int);
```

```

public java.util.Hashtable getQueryBandPairs(int, int[]);
public java.util.Hashtable getQueryBandPairs(java.lang.String,
    int, int[]);
public byte[] getQueryBandPairs(int);
public byte[] getQueryBandPairs(java.lang.String, int);
public int getQueryBandNumPairs(int);
public int getQueryBandNumPairs(java.lang.String, int);
public java.lang.String getQueryBandValue(int, java.lang.String);
public java.lang.String getQueryBandValue(java.lang.String,
    int, java.lang.String);
}

```

## getQueryBand(), getQueryBand(int *BufSize*)

Returns the current query band string for the transaction, session, and profile.

If the query band contains name-value pairs for the transaction, session, and/or profile, the method returns the concatenated transaction, session, and/or profile query band text.

### Syntax

```
getQueryBand(), getQueryBand(int BufSize)
```

### Syntax Elements

#### *BufSize*

[Optional] Specifies the maximum length of the query band string to return.

### Examples

If the query band contains name-value pairs for the transaction, session, and profile, the method returns the concatenated transaction, session, and profile query band text as follows:

```
=T> transaction_query_band =S> session_query_band =P> profile_query_band
```

Similarly, if the query band contains name-value pairs for the transaction and session, the method returns the concatenated transaction and session query band text as follows:

```
=T> transaction_query_band =S> session_query_band
```

If the query band contains name-value pairs for the transaction only, the text contains:

```
=T> transaction_query_band
```

If the query band contains name-value pairs for the session only, the text contains:

```
=S>  session_query_band
```

If the query band contains name-value pairs for the profile only, the text contains:

```
=P>  profile_query_band
```

## getQueryBandLen(int *BufSize*)

Returns the size in bytes of the query band.

### Syntax

```
getQueryBandLen(int BufSize)
```

### Syntax Elements

#### *BufSize*

Specifies the upper limit of the query band string.

## getQueryBandPairs(String *QBandBuf*, int *QB\_SearchType*)

Returns transaction, session, or profile name-value pairs from the query band string *QBandBuf*.

The encoding of the byte array is a name followed by a value, then a name followed by a value, and so forth, where, if the server character set is LATIN, a '\0' delimits each item and if the server character set is UNICODE, two '\0' delimit each item.

### Syntax

```
getQueryBandPairs(String QBandBuf, int QB_SearchType)
```

### Syntax Elements

#### *QBandBuf*

A query band string, where the query band can be:

- Returned by QueryBand.getQueryBand()
- Retrieved from the DBC.DBQLogTbl.QueryBand column
- Specified by the caller

**QB\_SearchType**

Specifies whether `getQueryBandPairs()` searches for name-value pairs in the transaction, session, and/or profile query band.

If the value is...

- `QueryBand.QB_FIRST`, then return the first unique name-value pair for each name found in the transaction, session, and profile query bands. If the transaction, session, and profile query bands contain the same name, `getQueryBandPairs()` returns the first name-value pair found in the query band in the following order:
  - Transaction query band
  - Session query band
  - Profile query band
- `QueryBand.QB_TXN`, then return name-value pairs in the transaction query band.
- `QueryBand.QB_SESSION`, then return name-value pairs in the session query band.
- `QueryBand.QB_PROFILE`, then return name-value pairs in the profile query band.

**getQueryBandNumPairs(String QBandBuf, int QB\_SearchType)**

Returns the number of name-value pairs in the query band that meet the search criteria specified by the `QB_SearchType` argument.

**Syntax**

```
getQueryBandNumPairs(String QBandBuf, int QB_SearchType)
```

**Syntax Elements****QBandBuf**

A query band string, where the query band can be:

- Returned by `QueryBand.getQueryBand()`
- Retrieved from the `DBC.DBQLogTbl.QueryBand` column
- Specified by the caller

**QB\_SearchType**

Specifies whether `getQueryBandNumPairs()` searches for name-value pairs in the transaction, session, and/or profile query band.

If the value is...

- `QueryBand.QB_FIRST`, then return the number of name-value pairs in the transaction, session, and profile query bands, where each name is unique. If the transaction,

session, or profile query bands contain the same name, `getQueryBandNumPairs()` counts only one name-value pair.

- `QueryBand.QB_TXN`, then return the number of name-value pairs in the transaction query band.
- `QueryBand.QB_SESSION`, then return the number of name-value pairs in the session query band.
- `QueryBand.QB_PROFILE`, then return the number of name-value pairs in the profile query band.

## **`getQueryBandValue(String QBandBuf, int QB_SearchType, String QBName)`**

Search the transaction, session, and/or profile name-value pairs in the query band string pointed to by the *QBandBuf* input argument and retrieve the value for a specified name.

### **Syntax**

```
getQueryBandValue(String QBandBuf, int QB_SearchType, String QBName)
```

### **Syntax Elements**

#### ***QBandBuf***

A query band string, where the query band can be:

- Returned by `QueryBand.getQueryBand()`
- Retrieved from the `DBC.DBQLogTbl.QueryBand` column
- Specified by the caller

#### ***QB\_SearchType***

Specifies whether `getQueryBandValue()` searches for name-value pairs in the transaction, session, and/or profile query band.

If the value is...

- `QueryBand.QB_FIRST`, then return the value of the first name-value pair, where the name is specified by the *QBName* input argument. If the query band string contains name-value pairs for the transaction, session, and profile, `getQueryBandValue()` searches the name-value pairs in the following order:
  - Transaction query band
  - Session query band
  - Profile query band



- QueryBand.QB\_TXN, then search the transaction name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.
- QueryBand.QB\_SESSION, then search the session name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.
- QueryBand.QB\_PROFILE, then search the profile name-value pairs in the query band and return the value that corresponds to the name specified by the *QBName* input argument.

***QBName***

The name in the name-value pair to return the value for.

**com.teradata.fnc.Runtime**

The following Java application classes support table operators.

**com.teradata.fnc.Runtime Classes**

- [Class ArrayTypeInfo](#)
- [Class ColumnDefinition](#)
- [Class InputInfo](#)
- [Class RuntimeContract](#)
- [Class StreamFormat](#)
- [Class UDTBaseInfo](#)

**com.teradata.fnc.Runtime Enums**

- [Enum InputInfo.StreamDir.](#)
- [Enum StreamFormat.FormatAttribute.](#)
- [Enum StreamFormat.StreamFmt.](#)

**Class ArrayTypeInfo**

The com.teradata.fnc.runtime.ArrayTypeInfo class contains Array metadata information for a column. This class corresponds to the C structure array\_info\_eon\_t used by C FNC functions. The Array metadata include the dimensions of the array, the number of elements, and information about each element.

The ArrayTypeInfo object is stored in the UDTBaseInfo class and can be retrieved using the getArrayInfo() method.

## ArrayTypeInfo Methods

The following methods retrieve Array information.

- `public int[][] getBounds()`
- `int getCharset()`
- `boolean getDefaultNull()`
- `TeradataType getElementType()`
- `int getMaxLength()`
- `int getNumDimensions()`
- `short getNumFractionalDigits()`
- `int getPeriodType()`
- `short getTotalIntervalDigits()`
- `int getTotalNumElements()`
- `short getUdtIndicator()`
- `java.lang.String getUdtName()`

The following methods set Array information.

- `void setBounds(int i, int j, int val)`
- `void setCharset(int charset)`
- `void setDefaultNull(boolean defaultNull)`
- `void setElementType(TeradataType type)`
- `void setMaxLength(int length)`
- `void setNumDimensions(int num)`
- `void setNumFractionalDigits(short digits)`
- `void setPeriodType(int pdttype)`
- `void setTotalIntervalDigits(short digits)`
- `void setTotalNumElements(int num)`
- `void setUdtIndicator(short indicator)`
- `void setUdtName(java.lang.String udtname)`

## Class ColumnDefinition

```
public class ColumnDefinition
extends java.lang.Object
```

The ColumnDefinition class provides a means for the user to define the output columns.

## ColumnDefinition Constructors

After the class is instantiated, the characteristics of the type can be set using the methods `setPrecision`, `setScale`, `setDisplayLength`, and so on. Calling these methods for types that do not support the specific method results in an exception. The `com.teradata.fnc.TeradataType` permits access to Teradata specific types.

### Syntax

```
public ColumnDefinition(java.lang.String Name,
                        int typeval)
```

### Syntax Elements

#### *Name*

The name of the output column.

#### *typeval*

The `java.sql.Type` of the column.

### Syntax

```
public ColumnDefinition(java.lang.String Name,
                        com.teradata.fnc.TeradataType Type)
```

### Syntax Elements

#### *Name*

The name of the output column.

#### *typeval*

The `com.teradata.fnc.TeradataType` of the column.

## ColumnDefinition Methods

### Inherited Methods

The following methods are inherited from class `java.lang.Object`:

- `equals`
- `getClass`
- `hashCode`
- `notify`

- notifyAll
- toString
- wait, wait, wait

## getCharset()

Provides access to the character set of the column.

### Syntax

```
public int getCharset()
```

## getDisplayLength()

Provides access to the length of the data type of the column.

### Syntax

```
public int getDisplayLength()
```

## getJsonFormat()

Provides access to the JSON storage format of the JSON column.

### Syntax

```
public byte getJsonFormat()
```

## getName()

Returns a string with the column name.

### Syntax

```
public java.lang.String getName()
```

## getPeriodType()

Returns an int value indicating the Period type of the column. Valid values are as follows:

```

NOT_PERIOD = 0
  PERIOD_DATE = 1
  PERIOD_TIME = 2
  PERIOD_TIME_WTZ = 3
  PERIOD_TIMESTAMP = 4
  PERIOD_TIMESTAMP_WTZ = 5

```

**Syntax**

```
public int getPeriodType()
```

**getPrecision()**

Provides access to the current precision for a DECIMAL column.

**Syntax**

```
public int getPrecision()
```

**getScale()**

Provides access to the scale for a DECIMAL column.

**Syntax**

```
public int getScale()
```

**getType()**

Returns the Teradata type of the column.

**Syntax**

```
public com.teradata.fnc.TeradataType getType()
```

**getUDTName()**

Returns a String that is the UDT name of the column.

**Syntax**

```
public String getUdtName()
```

**isTimeZone()**

Provides access to the time zone of a TIME or DATE type column.

**Syntax**

```
public boolean isTimeZone()
```

**setCharset(int)**

Sets the character set of the column.

**Syntax**

```
public void setCharset(int cset)
```

**Syntax Elements*****cset***

- A value of 1 specifies a LATIN column.
- A value of 2 specifies a UNICODE column.

**setDisplayLength(int)**

Sets the length of a variable-length data type column. You can use this method for CHARACTER, VARCHAR, CLOB, VARBYTE, and BLOB types.

**Syntax**

```
public void setDisplayLength(int len)
```

**Syntax Elements*****len***

The length of a variable-length data type column.

## setJsonFormat(byte)

Sets the JSON storage format of a JSON column.

### Syntax

```
public void setJsonFormat(byte p)
```

## setPeriodType(int)

Sets the Period type of the column. Valid values are as follows:

```
NOT_PERIOD = 0
  PERIOD_DATE = 1
  PERIOD_TIME = 2
  PERIOD_TIME_WTZ = 3
  PERIOD_TIMESTAMP = 4
  PERIOD_TIMESTAMP_WTZ = 5
```

### Syntax

```
public void setPeriodType(int ptype)
```

## setPrecision(int)

Sets the precision for a DECIMAL column.

### Syntax

```
public void setPrecision(int prec)
```

### Syntax Elements

*prec*

The precision for the DECIMAL column.

## setScale(int)

Sets the scale for a DECIMAL column.

**Syntax**

```
public void setScale(int Scale)
```

**Syntax Elements*****Scale***

The scale for the DECIMAL column.

**setTimeZone(boolean)**

Sets the time zone of a TIME or DATE type column.

**Syntax**

```
public void setTimeZone(boolean isWTZ)
```

**Syntax Elements*****isWTZ***

The time zone for the column.

**setUDTName(String)**

Sets the UDT name of the output column.

**Syntax**

```
public void setUDTName(String name)
```

**Class InputInfo**

Class InputInfo provides methods to retrieve the metadata for a table operator. InputInfo provides access to the Map containing the Custom clause information and can use standard Map methods to retrieve items by index or name. Similarly, the columns that participate in the Hash or Ordering of the input are available from lists with the getHashBy and getOrderBy methods. A Map of function information is also available so you can retrieve the function name and any other attributes.

**Syntax**

```
public class InputInfo  
extends java.lang.Object
```



## InputInfo Methods

### Inherited Methods

The following methods are inherited from class `java.lang.Object`:

- `equals`
- `getClass`
- `hashCode`
- `notify`
- `notifyAll`
- `toString`
- `wait, wait, wait`

### getAsName(int)

Returns the name associated with an input stream.

#### Syntax

```
public String getAsName(int StreamNo)
```

#### Syntax Elements

##### *StreamNo*

The stream number.

### getCustom()

Returns a `Map` object containing the custom clause information for the table operator in the form of name-value pairs. If there is no custom clause specified for the table operator, the method returns a non-null empty `Map` object.

#### Syntax

```
public java.util.Map<java.lang.String,java.lang.Object> getCustom()
```

### getFormat(int, InputInfo.StreamDir)

Returns a `Map` of the format settings.

**Syntax**

```
public java.util.Map<StreamFormat.FormatAttribute,java.lang.Object>
    getFormat(int streamno, InputInfo.StreamDir dir)
```

**Syntax Elements*****StreamNo***

The stream number.

***dir***

The direction of the stream (input or output).

**getFunctionInfo()**

Returns a Map of the function information. This includes the function name, the execution mode, case sensitivity, and character type.

**Syntax**

```
public java.util.Map<java.lang.String,java.lang.Object>
    getFunctionInfo()
```

**getHashBy(int)**

Returns a List of the HASH BY columns (names only).

**Syntax**

```
public java.util.List<ColumnDefinition> getHashBy(int StreamNo)
```

**Syntax Elements*****StreamNo***

The stream number.

**getIncount()**

Returns the number of input streams.

**Syntax**

```
public int getIncount()
```

**getisDimension(int)**

Returns true if the input stream is a DIMENSION input; otherwise, it returns false.

**Syntax**

```
public boolean  getisDimension(int  StreamNo)
```

**Syntax Elements*****StreamNo***

The stream number.

**getOrderBy(int)**

Returns a List of the ORDER BY columns (names only).

**Syntax**

```
public java.util.List<ColumnDefinition> getOrderBy(int  StreamNo)
```

**Syntax Elements*****StreamNo***

The stream number.

**getOutcount()**

Returns the number of output streams.

**Syntax**

```
public int getOutcount()
```

## getUDTMetadata(int)

Retrieves the UDT metadata for a particular stream. The method returns an array of UDTBaseInfo objects that contain the UDT metadata for each column in the input stream.

### Syntax

```
UDTBaseInfo[] getUDTMetadata(int StreamNo)
```

### Syntax Elements

#### *StreamNo*

The stream number.

## Enum InputInfo.StreamDir

```
public static enum InputInfo.StreamDir
extends java.lang.Enum<InputInfo.StreamDir>
```

All implemented interfaces:

- java.io.Serializable
- java.lang.Comparable<InputInfo.StreamDir>

Enclosing class: InputInfo

## Enum InputInfo.StreamDir Constants

```
public static final InputInfo.StreamDir StreamIn
```

```
public static final InputInfo.StreamDir StreamOut
```

## Enum InputInfo.StreamDir Methods

### Inherited Methods

The following methods are inherited from class java.lang.Enum:

- compareTo
- equals
- getDeclaringClass
- hashCode
- name

- ordinal
- toString
- valueOf

The following methods are inherited from class `java.lang.Object`:

- getClass
- notify
- notifyAll
- wait, wait, wait

## getStreamFormat

### Syntax

```
public int getStreamFormat()
```

## valueOf

Returns the enum constant of this type with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. Extraneous whitespace characters are not permitted.

### Syntax

```
public static InputInfo.StreamDir valueOf(java.lang.String name)
```

### Syntax Elements

#### *name*

the name of the enum constant to be returned.

### Exceptions

`valueOf` throws the following exceptions:

- `java.lang.IllegalArgumentException` - if this enum type has no constant with the specified name.
- `java.lang.NullPointerException` - if the argument is null.

## values

Returns an array containing the constants of this enum type in the order they are declared. This method may be used to iterate over the constants as follows:

```
for (InputInfo.StreamDir c : InputInfo.StreamDir.values())
    System.out.println(c);
```

### Syntax

```
public static InputInfo.StreamDir[] values()
```

## Class RuntimeContract

Container for all runtime information.

### Syntax

```
public class RuntimeContract
    extends java.lang.Object
```

## RuntimeContract Methods

### Inherited Methods

The following methods are inherited from class `java.lang.Object`:

- `equals`
- `getClass`
- `hashCode`
- `notify`
- `notifyAll`
- `toString`
- `wait, wait, wait`

## bytesTransferred

Records the number of bytes transferred between Vantage and the foreign server by the table operator. This routine is callable on an AMP vproc only by a table operator.

### Syntax

```
public void bytesTransferred(long in, long out)
    throws SQLException
```

## Syntax Elements

*in*

The number of bytes transferred into Vantage from the foreign server.

*out*

The number of bytes transferred from Vantage to the foreign server.

## complete

Finalizes the contract with the Teradata parser.

### Syntax

```
public void complete()
```

## disableCoGroup

Disables the cogroup functionality for table operators that handle multiple input streams.

### Note:

If cogroup is disabled, table operators that handle multiple input streams may return different results on systems with different configurations where the number of AMPs differ. To get consistent results on different configurations, cogroup must be enabled.

### Syntax

```
public void disableCoGroup()
```

### Example

```
public int contract(RuntimeContract contract, ResultSet rsin[],
    ResultSet rsout[])
    throws SQLException
{
    DbsInfo.traceWrite("MITblOpContract");
    contract.disableCoGroup();
    ...
}
```

## getAmpHash

Returns values that hash to the specified AMPs. This routine is callable on an AMP or PE vproc.

### Syntax

```
public void getAmpHash(int[][] amphash)
```

### Syntax Elements

#### *amphash*

*amphash*[*n*][0] is the AMP number.

*amphash*[*n*][1] will be returned with the value that hashes to the AMP.

*n* is the size of the *amphash* array.

## getBaseInfo

Retrieves metadata information about a UDT or CDT column.

The method returns an array of UDTBaseInfo objects for the ColumnDefinition objects passed in. This method can only be invoked in the contract function.

### Syntax

```
public UDTBaseInfo[] getBaseInfo(ColumnDefinition[] colDefs)
    throws SQLException
```

### Syntax Elements

#### *colDefs*

A list of column definitions.

## getContractCtx

Provides access to the contract bytes in the AMP. This method is used with setContractCtx(byte []).

byte[] returns the object from setContractCtx(Object) from the Teradata parser to the operator in the AMP.

### Syntax

```
public byte[] getContractCtx()
```



## getContractObject

Provides access to the serialized contract object in the AMP. This method is used with setContracCtx(Object).

Object returns the object from setContracCtx(Object) from the Teradata parser to the operator in the AMP.

### Syntax

```
public java.lang.Object  getContractObject()
    throws java.io.IOException,
           java.lang.ClassNotFoundException
```

## getContractPhase

Indicates the phase in the parser from which the contract function is being called. The parser phases are as follows:

Parser Phase	Meaning
FNC_CTRCT_GET_ALLCOLS_PHASE = 0	Function returns all remote table columns.
FNC_CTRCT_VALIDATE_PHASE = 1	Validates that inputs are correct. Contract function can be called multiple times from this phase. <b>Note:</b> This phase is currently not used.
FNC_CTRCT_COMPLETE_PHASE = 2	Last call of contract function. Necessary foreign server actions must be completed.
FNC_CTRCT_DDL_PHASE = 3	CREATE SERVER statement execution is being completed and connectivity must be verified.
FNC_CTRCT_DEFINE_SERVER_PHASE = 4	CREATE VIEW or CREATE MACRO statement is being executed. Custom clause data may be invalid.

This routine is callable on a PE vproc only by a table operator.

### Syntax

```
public ContractPhase  getContractPhase();
```

## getDBSSessionAttrInfo

Returns information about the current session to a table operator. `getDBSSessionAttrInfo` returns a string in JSON format containing current session information, such as system variables and session attributes, including the following:

- Current user name
- Current role name
- Transaction mode
- Collation

### Syntax

```
String getDBSSessionAttrInfo();
```

### Usage Notes

For current role name, a maximum of 127 role names are returned. If a user is granted more than 127 roles and the user issues a SET ROLE ALL statement to enable all of the roles, "ALL" is returned for the current role name. For example, the following sample output shows the session information returned for TESTUSER who has more than 127 roles enabled:

```
{"CurUserName": "TESTUSER", "CurRoleName": "ALL", "TransactionMode": "BTET", "Collatio  
n": "ASCII"}
```

For users who specify SET ROLE ALL but have 127 roles or less, "ALL" is returned along with each of the role names. For example, TESTUSER2 has 3 roles: Role1, Role2, and Role3.

```
{"CurUserName": "TESTUSER2", "CurRoleName":  
["ALL", "Role1", "Role2", "Role3"], "TransactionMode": "BTET", "Collation": "ASCII"}
```

## getExternalQuery

Generates the text query string for the foreign server and returns the interface version that is currently supported. The method returns a string which contains the external query.

This routine is callable on a PE vproc only by a table operator.

### Syntax

```
public String getExternalQuery(  
    ColumnDefinition[] colDefs,  
    ServerType serverType,
```

```
ExtOpSetType[]      extOpSetTypes,
int[]               interfaceVersions)
    throws SQLException
```

## Syntax Elements

### *colDefs*

A list of column definitions.

### *serverType*

ServerType is defined as follows:

```
public enum ServerType {
    ANSISQL(0),
    HADOOP(1);
}
```

### *extOpSetTypes*

A set of valid operators supported on the foreign server.

ExOpSetType is defined as follows:

```
public enum ExtOpSetType {
    Eq_ET(0),
    Ne_ET(1),
    Gt_ET(2),
    Le_ET(3),
    Lt_ET(4),
    And_ET(5),
    Or_ET(6),
    Not_ET(7),
    Between_ET(8),
    In_ET(9),
    NotIn_ET(10),
    Ge_ET(11),
    Like_ET(12),
    LastOp_ET(13);
}
```

### *interfaceVersions*

- The caller passes in the desired interface version as the argument.

- The routine returns the actual interface version that is currently supported.

### Example: Calling `getExternalQuery`

```
ServerType sType = ServerType.ANSISQL;
ExtOpSetType extOpTypes[] = new ExtOpSetType[3];
extOpTypes[0] = ExtOpSetType.Eq_ET;
extOpTypes[1] = ExtOpSetType.And_ET;
extOpTypes[2] = ExtOpSetType.Or_ET;

int[] versions = new int[2];
versions[0] = 1;    // The caller passes in the desired interface version.

String extQuery = contract.getExternalQuery(colDefs, sType,
extOpTypes, versions);
```

After calling `getExternalQuery`, `versions[1]` will contain the actual interface version that is currently supported on the system.

## getHashAmp

Accepts data and determines the AMP which would be responsible for that key. The method returns an integer representing the number of the AMP that would be responsible for the key. This routine is callable on a PE vproc only by a table operator.

### Syntax

```
public int  getHashAmp(Object[]  data)
```

### Syntax Elements

#### *data*

An array representing table columns.

## getInnerContractCtx

Gets the contract definition of a nested inner table operator for the outer table operator to use. NULL is returned if the inner contract function does not exist. This routine is callable on a PE vproc only by a table operator.

**Syntax**

```
public byte[]  getInnerContractCtx()
    throws SQLException
```

**getInputInfo**

Provides access to the InputInfo container to get metadata about the execution of the table operator.

**Syntax**

```
public InputInfo  getInputInfo()
```

**getUniqTblOpID**

Returns the unique identifier associated with a table operator.

When multiple table operators are present in a single query, as in a map reduce operation such as SELECT \* FROM operator (on (select \* from operator), you can use the unique table identifier to distinguish between each of the table operators.

**Syntax**

```
public int  getUniqTblOpID()
```

**setActivityCount**

Sets the number of rows exported. This routine is callable on an AMP vproc only by a table operator

**setActivityCount**

```
public void  setActivityCount(int  stream, long  rowsexported)
    throws SQLException
```

**Syntax Elements*****stream***

Specifies which stream to write to.

***rowsexported***

The value to be written to ActivityCount.

## setContractCtx

Defines a byte [] to be passed from the Teradata parser to the AMPs.

### Syntax

```
public void setContractCtx(byte[] ctx)
    throws java.sql.SQLException
```

### Syntax Elements

*ctx*

The data to be passed to the AMPs.

## setContractCtx

Serializes and sets the contract object. Class object must support writeObject().

### Syntax

```
public void setContractCtx(java.lang.Object Obj)
    throws java.io.IOException
```

### Syntax Elements

*Obj*

The contract object.

## setDisplayLength

Resets the lengths in column definitions for VARCHAR data types. This routine can be invoked for both import and export operations.

The routine is callable on a PE vproc only by a table operator.

### Syntax

```
public void setDisplayLength(char direction,
                             ColumnDefinition[] colDefs)
    throws SQLException
```

## Syntax Elements

### *direction*

IN parameter. Specify an input value of 'R' for export and 'W' for import.

### *colDefs*

IN/OUT parameter. The column definitions for which the display lengths will be reset.

## setExplainText

Sets the EXPLAIN text when the table operator has the hexplain custom clause set.

Hexplain has the following values for the type of EXPLAIN to be completed:

- 1 = simple
- 2 = verbose
- 3 = DBQL

This routine accepts multiple self-contained EXPLAIN text strings as input in order to handle a multi-row EXPLAIN plan from a foreign server. The routine provides the EXPLAIN plan to the parser which will display the multiple lines of the EXPLAIN plan.

This routine is callable on a PE vproc only by a table operator.

## Syntax

```
public void setExplainText(String[] texts);
```

## Syntax Elements

### *texts*

An array containing the EXPLAIN text strings.

## setFormat

Sets attributes of the format of the input and output streams. This allows the contract function to specify the format of the data types to the parser.

## Syntax

```
public void setFormat(
    int stream,
    InputInfo.StreamDir dir,
    java.util.Map<StreamFormat.FormatAttribute, java.lang.Object> formatattributes)
```

## Syntax Elements

### *stream*

IN parameter. Indicates the stream on which the format will be applied. Currently the only valid value is 0.

### *dir*

IN parameter. The direction of the stream (input or output).

### *formatattributes*

IN parameter. Map of attribute values to apply. Valid attributes are as follows:

- "RECFMT"
- "TZTYPE"
- "CHARSETFMT"
- "REPUNSPTCHR"

"CHARSETFMT" and "REPUNSPTCHR" apply only to import table operators.

## Usage Notes

- This routine is valid only when called within the contract function of a table operator.
- For "RECFMT" the default value is INDICFMT1, where the format is IndicData with row separator sentinels. All field-level formats impact the entire record.
- If data being imported from a foreign server contains unsupported characters, you must use setFormat() and explicitly set "CHARSETFMT" and "REPUNSPTCHR" attributes.
- Format Attribute Values:

Parameter Name	Definition
"RECFMT"	Defines the record format: <ul style="list-style-type: none"> <li>◦ INDICFMT1 = 1 IndicData with row separator sentinels.</li> <li>◦ INDICBUFFMT1 = 2 IndicData with NO row or partition separator sentinels.</li> </ul>
"TZTYPE"	Used as an indicator to Vantage to receive from or send TIME/TIMESTAMP data to the table operator in a different format. <ul style="list-style-type: none"> <li>◦ RAW = 0 as stored on the Teradata file system</li> <li>◦ UTC = 1 as UTC</li> </ul>
"CHARSETFMT"	<ul style="list-style-type: none"> <li>◦ EVLDBC Signals that neither data conversion nor detection is needed.</li> <li>◦ EVLUTF16CHARSET Signals that the external data to be imported into Vantage are in UTF16 encoding.</li> </ul>



Parameter Name	Definition
	<ul style="list-style-type: none"> <li>◦ EVLUTF8CHARSET Signals that the external data to be imported into Vantage are in UTF8 encoding.</li> </ul>
"REPUNSPTCHR"	<p>A boolean value that specifies what to do when an unsupported unicode character is detected in the external data to be imported into Vantage.</p> <ul style="list-style-type: none"> <li>◦ true Replaces the unsupported character with U+FFFD.</li> <li>◦ false Return an error when an unsupported character is detected. This is the default behavior.</li> </ul>

- Importing and Exporting TIME/TIMESTAMP Data

You can map the Teradata TIME and TIMESTAMP data types to the Hadoop STRING or the Oracle TIMESTAMP data type when importing or exporting data to these foreign servers.

The table operator can use `setFormat()` to set the `tztype` attribute as an indicator to Vantage to receive from or send TIMESTAMP data to the table operator in a native but adjusted format.

The `tztype` attribute is set as follows for the import and export operators:

- For Hadoop, the attribute is set to UTC.
- For Oracle, the attribute is set to UTC.

If the transform is off, the data will be transferred in Raw form which is the default for table operators and is consistent with standard UDFs.

`tztype` is a member of the structure `FNC_FmtConfig_t` defined in `fnctypes.h` as follows:

```
typedef struct
{
    int Stream_Fmt_en recordfmt; //enum - indicdata, fastload
    binary, delimited
    bool inlinelob; //inline or deferred
    bool UDTTransformsOff; //true or false
    bool PDTTransformsOff; //true or false
    bool ArrayTransformsOff; //true or false
    char auxinfo[128]; //For delimited text can contain the record
    separator, delimiter
    //specification and the field enclosure characters
    double inperc; //recommended percentage of buffer devoted to input rows
    bool inputnames; //send input column names to step
    bool outputnames; //send output column names to step
    TZType_en tztype; //enum - Raw or UTC
    int charsetfmt; // charset format of data to be imported into TD
    through QG
}
```

```

    bool replUnsprtedUniChar; /* true - replace unsupported unicode character
                               encountered with U+FFFD when data
is imported
                               into TD through QG
                               false - error out when unsupported unicode
                               char encountered */
} FNC_FmtConfig_t;

```

TZType\_en is defined as follows:

```

typedef enum
{
    Raw = 0, /* as stored on TD File system */
    UTC = 1, /* as UTC */
} TZType_en;

```

For export, setInputInfo() is called during the contract phase in the resolver and will use the tztype attribute to add the desired cast to the input TIME or TIMESTAMP column types.

Vantage converts the TIME and TIMESTAMP data to the session local time before casting to the character type, so when a TIME or TIMESTAMP column is being mapped to charfix/charvar as when mapping to the Hadoop STRING type, the data will transmit in session local time zone and no explicit casts are needed.

For import, when getting the input buffer from the table operator, TIME or TIMESTAMP data have to be converted to Raw form. There is no conversion needed for the import of Hadoop Strings to Vantage TIME or TIMESTAMP data types since it follows the normal conversion path from character to TIME/TIMESTAMP in Vantage.

---

#### Note:

Teradata does not recommend importing or exporting TIME/TIMESTAMP data for a Teradata system with timedatewzcontrol flag 57 = 0. For such systems, the TIME/TIMESTAMP data is stored in OS local time. The System/Session time zone is not set and Vantage does not apply any conversions on TIME/TIMESTAMP data when reading or writing from disk. Therefore, exporting such data reliably in the format desired by the foreign server is a problem and Teradata recommends that the Teradata-to-Hadoop connector feature not be used for such systems.

---

## setHashBy

Allows the contract function writer to set the HASH BY specification when developing table operators.

This routine can only run if called from the contract function. It is callable on a PE vproc.

The routine will produce an error if the stream number is invalid or the HASH BY metadata was already set.

**Syntax**

```
public void setHashBy(int streamno,
                     String[] colNames)
    throws SQLException
```

**Syntax Elements*****streamno***

The input stream number.

***colNames***

A pointer to the HASH BY metadata.

**setInputInfo**

Sets casting statements on the input columns so that the data types are cast as indicated by the caller.

The routine is callable on a PE vproc only by a table operator.

**Syntax**

```
public void setInputInfo(int streamno,
                        ColumnDefinition[] colDefs)
    throws SQLException
```

**Syntax Elements*****streamno***

The input stream number.

***colDefs***

A list of column definitions.

**setOrderBy**

Allows the contract function writer to set the ordering specification when developing table operators.

This routine can only run if called from the contract function. It is callable on a PE vproc.

The routine will produce an error if the stream number is invalid or the LOCAL ORDER BY metadata was already set.

**Syntax**

```
public void setOrderBy(int streamno,
                       String[] colNames)
    throws SQLException
```

**Syntax Elements*****streamno***

The input stream number.

***colNames***

A pointer to the LOCAL ORDER BY metadata.

**setOutputInfo**

Defines the output information for the stream.

**Syntax**

```
public void setOutputInfo(int streamno,
                           ColumnDefinition[] info)
    throws java.sql.SQLException
```

**Syntax Elements*****streamno***

Indicates the stream to which the column definitions will be applied. Currently the only valid value is 0.

***info***

The defined columns.

**Class StreamFormat**

Class StreamFormat defines the format information for each stream.

**Syntax**

```
public class StreamFormat
    extends java.lang.Object
```

## StreamFormat Constructor

Defines the format options for an output stream.

### Syntax

```
public StreamFormat(int stream,
                    InputInfo.StreamDir Dir,
                    java.util.Map<StreamFormat.FormatAttribute, java.lang.Object> map)
```

### Syntax Elements

#### *stream*

Indicates the stream on which the format will be applied. Currently the only valid value is 0.

#### *Dir*

The direction of the stream (input or output).

#### *map*

Map of attribute values to apply.

## StreamFormat Methods

### Inherited Methods

The following methods are inherited from class `java.lang.Object`:

- equals
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait, wait, wait

### getAttributeMap

Provides access to the format attributes for an output stream.

getAttributeMap returns a Map of attributes.

**Syntax**

```
public
java.util.Map<StreamFormat.FormatAttribute,java.lang.Object>  getAttributeMap()
```

**Enum StreamFormat.FormatAttribute**

All implemented interfaces:

- java.io.Serializable
- java.lang.Comparable<StreamFormat.FormatAttribute>

Enclosing class: StreamFormat

**Syntax**

```
public static enum  StreamFormat.FormatAttribute
extends java.lang.Enum<StreamFormat.FormatAttribute>
```

**Enum StreamFormat.FormatAttribute Constants****Syntax**

```
public static final StreamFormat.FormatAttribute  ARRAYXFORMS
```

```
public static final StreamFormat.FormatAttribute  AUXINFO
```

```
public static final StreamFormat.FormatAttribute  INLINELOB
```

```
public static final StreamFormat.FormatAttribute  INNAME
```

```
public static final StreamFormat.FormatAttribute  INPERC
```

```
public static final StreamFormat.FormatAttribute  OUTNAME
```

```
public static final StreamFormat.FormatAttribute  PDTXFORMS
```

```
public static final StreamFormat.FormatAttribute  RECFMT
```

```
public static final StreamFormat.FormatAttribute  UBFMTATT
```

```
public static final StreamFormat.FormatAttribute  UDTXFORMS
```

## Enum StreamFormat.FormatAttribute Methods

### Inherited Methods

The following methods are inherited from class `java.lang.Enum`:

- `compareTo`
- `equals`
- `getDeclaringClass`
- `hashCode`
- `name`
- `ordinal`
- `toString`
- `valueOf`

The following methods are inherited from class `java.lang.Object`:

- `getClass`
- `notify`
- `notifyAll`
- `wait, wait, wait`

### getFormatAttribute

#### Syntax

```
public int getFormatAttribute()
```

### size

#### Syntax

```
public int size()
```

### valueOf

Returns the enum constant of this type with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. Extraneous whitespace characters are not permitted.

#### Syntax

```
public static StreamFormat.FormatAttribute valueOf(java.lang.String name)
```

## Syntax Elements

*name*

the name of the enum constant to be returned.

## Exceptions

valueOf throws the following exceptions:

- java.lang.IllegalArgumentException - if this enum type has no constant with the specified name.
- java.lang.NullPointerException - if the argument is null.

## values

Returns an array containing the constants of this enum type in the order they are declared. This method may be used to iterate over the constants as follows:

```
for (StreamFormat.FormatAttribute c : StreamFormat.FormatAttribute.values())
    System.out.println(c);
```

## Syntax

```
public static StreamFormat.FormatAttribute[] values()
```

## Enum StreamFormat.StreamFmt

All implemented interfaces:

- java.io.Serializable
- java.lang.Comparable<StreamFormat.StreamFmt>

Enclosing class: StreamFormat

## Syntax

```
public static enum StreamFormat.StreamFmt
extends java.lang.Enum<StreamFormat.StreamFmt>
```

## Enum StreamFormat.StreamFmt Constants

### Syntax

```
public static final StreamFormat.StreamFmt INDICBUFFMT1
```



```
public static final StreamFormat.StreamFmt INDICFMT1
```

## Enum StreamFormat.StreamFmt Methods

### Inherited Methods

The following methods are inherited from class java.lang.Enum:

- compareTo
- equals
- getDeclaringClass
- hashCode
- name
- ordinal
- toString
- valueOf

The following methods are inherited from class java.lang.Object:

- getClass
- notify
- notifyAll
- wait, wait, wait

## getStreamFormat

### Syntax

```
public int getStreamFormat()
```

### valueOf

Returns the enum constant of this type with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. Extraneous whitespace characters are not permitted.

### Syntax

```
public static StreamFormat.StreamFmt valueOf(java.lang.String name)
```

## Syntax Elements

*name*

the name of the enum constant to be returned.

## Exceptions

valueOf throws the following exceptions:

- java.lang.IllegalArgumentException - if this enum type has no constant with the specified name.
- java.lang.NullPointerException - if the argument is null.

## values

Returns an array containing the constants of this enum type in the order they are declared. This method may be used to iterate over the constants as follows:

```
for (StreamFormat.StreamFmt c : StreamFormat.StreamFmt.values())
    System.out.println(c);
```

## Syntax

```
public static StreamFormat.StreamFmt[] values()
```

## Class UDTBaseInfo

The com.teradata.fnc.runtime.UDTBaseInfo class provides a means for the user to provide metadata about UDTs and complex types (CDTs). It corresponds to the C UDT\_BaseInfo\_t structure used by C FNC functions. For information about this structure, see [Table Operator Data Structures](#).

## UDTBaseInfo Methods

### getArrayInfo()

Retrieves all Array type related information and returns it in the ArrayTypeInfo object. The ArrayTypeInfo class corresponds to the array\_info\_eon\_t structure used by C FNC functions. This structure contains information about the dimensions of the array, the number of elements, and information about each element. For information about the array\_info\_eon\_t structure, see the sqltypes\_td.h header file.

## Syntax

```
public ArrayTypeInfo getArrayInfo()
```

## getArrayNumDimension()

Retrieves the number of dimensions for an Array/Varray type. This method is not applicable to any other UDT or complex types.

The method returns a constant of type `int` that contains the number of dimensions of this Array/Varray type.

The method throws `SQLException` if an error occurs while attempting to access the dimensions information. A database specific code “9744 (ERRUDFJAVUDT) < Number of dimensions for Array/Varray could not be accessed>” is returned.

### Syntax

```
int getArrayNumDimension()
    throws SQLException
```

## getBaseCharset()

Retrieves the character set for the UDT or complex type, if applicable to its base type.

The method returns a constant of type `int` that contains the charset value.

The method throws `SQLException` if an error occurs while attempting to access the charset value. A database specific code “9744 (ERRUDFJAVUDT) <UDT charset could not be accessed>” is returned.

### Syntax

```
int getBaseCharset()
    throws SQLException
```

## getBaseDataType()

Retrieves the base data type information for the UDT or complex type. Valid values include those contained in the Java enum `TeradataType`.

The method returns a member of the enum `TeradataType` that matches the base type of this UDT or complex type.

The method throws `SQLException` if an error occurs while attempting to access the base data type. A database specific code “9744 (ERRUDFJAVUDT) <UDT base type could not be accessed>” is returned.

**Syntax**

```
TeradataType getBaseDataType()
               throws SQLException
```

**getBaseMaxLength()**

Retrieves the maximum length for the base type of the UDT or complex type.

The method returns a constant of type `int` that contains the base maximum length value for the UDT or complex type.

The method throws `SQLException` if an error occurs while attempting to access the base maximum length. A database specific code “9744 (ERRUDFJAVUDT) <UDT base max length could not be accessed >” is returned.

**Syntax**

```
int getBaseMaxLength()
    throws SQLException
```

**getBaseNumFractionalDigits()**

Retrieves the fractional digits value for the base type of the UDT or complex type. This value corresponds to the value  $n$  in a `DECIMAL( $m,n$ )` base type; otherwise it is set to zero.

The method returns a constant of type `short` that contains the fractional digits value.

The method throws `SQLException` if an error occurs while attempting to access the total interval digits value. A database specific code “9744 (ERRUDFJAVUDT) <UDT base fractional digits could not be accessed >” is returned.

**Syntax**

```
short getBaseNumFractionalDigits()
      throws SQLException
```

**getBasePrecision()**

Retrieves the precision for the base type of the UDT or complex type. This value corresponds to the precision for time or timestamp types; for example `TIME(4)` or `TIMESTAMP(6)`, otherwise it is set to zero.

The method returns a constant of type `int` that contains the precision value.

The method throws `SQLException` if an error occurs while attempting to access the precision value. A database specific code “9744 (ERRUDFJAVUDT) <UDT base precision could not be accessed>” is returned.

### Syntax

```
int getBasePrecision()
    throws SQLException
```

## getBaseTotalIntervalDigits()

Retrieves the total interval digits for the base type of the UDT or complex type. This value corresponds to the value  $m$  in a `DECIMAL( $m,n$ )` base type; otherwise it is set to zero.

The method returns a constant of type `short` that contains the total interval digits value.

The method throws `SQLException` if an error occurs while attempting to access the total interval digits value. A database specific code “9744 (ERRUDFJAVUDT) <UDT base total interval digits could not be accessed>” is returned.

### Syntax

```
short getBaseTotalIntervalDigits()
    throws SQLException
```

## getJSONStorageFormat()

Retrieves the JSON storage format for a JSON column. Valid values match those used for the C structure `json_storage_en`. For information about this structure, see [Table Operator Data Structures](#).

The method returns a constant of type `short` that contains the JSON storage format value corresponding to this JSON column.

The method throws `SQLException` if an error occurs while attempting to access the JSON storage format value. A database specific code “9744 (ERRUDFJAVUDT) <JSON Storage format could not be accessed>” is returned.

### Syntax

```
short getJSONStorageFormat()
    throws SQLException
```

## getStructNumAttributes()

Retrieves the number of total attributes for a structured UDT. This method is not applicable to any other UDT or complex types.

The method returns a constant of type `int` that contains the number of attributes for this structured UDT.

The method throws `SQLException` if an error occurs while attempting to access the number of attributes of the structured UDT. A database specific code “9744 (ERRUDFJAVUDT) <Structured UDT number of attributes could not be accessed>” is returned.

### Syntax

```
int getStructNumAttributes()
    throws SQLException
```

## getTransform Methods

Information about the transform type of a UDT can be retrieved using the following methods:

- `int getTransformCharset()`
- `int getTransformFractionalDigits()`
- `int getTransformIntervalRange()`
- `int getTransformMaxLength()`
- `int getTransformPrecision()`
- `int getTransformTotalDigits()`
- `TeradataType getTransformType()`

These methods are similar to the `getBase` type methods but refer to the default transform type of the UDT.

## getUdtIndicator()

Retrieves the UDT indicator attribute for the UDT or CDT.

Valid values match those used for the `udt_indicator` attribute of the C structure `UDT_BaseInfo_t`. For information about this structure, see [Table Operator Data Structures](#).

The method returns a constant of type `short` that contains the indicator value corresponding to this UDT or CDT.

The method throws `SQLException` if an error occurs while attempting to access the indicator value. A database specific code “9744 (ERRUDFJAVUDT) <UDT indicator could not be accessed>” is returned.

**Syntax**

```
short getUdtIndicator()
           throws SQLException
```

**getUdtName()**

Retrieves the name for a UDT or complex type.

The method returns a constant of type `java.lang.String` that contains the name of the UDT.

The method throws `SQLException` if an error occurs while attempting to access the `udtName`. A database specific code “9744 (ERRUDFJAVUDT) <UDT name could not be accessed>” is returned.

**Syntax**

```
java.lang.String getUdtName()
           throws SQLException
```

**setArrayInfo(ArrayTypeInfo)**

Sets the Array metadata information for a column. The information that is set includes the dimensions of the array, the number of elements, and information about each element.

**Syntax**

```
public void setArrayInfo(ArrayTypeInfo a)
```

**setArrayNumDimension(int)**

Sets the number of dimensions for an Array/Varray type. This method is not applicable to any other UDT or complex types.

The method throws `SQLException` if an error occurs while attempting to set the number of dimensions for an Array/Varray type. A database specific code “9744 (ERRUDFJAVUDT) <Number of dimensions for Array/Varray could not be set>” is returned.

**Syntax**

```
void setArrayNumDimension(int num)
           throws SQLException
```

## setBaseCharset(int)

Sets the character set for the UDT or complex type, if applicable to its base type.

The method throws `SQLException` if an error occurs while attempting to set the charset. A database specific code “9744 (ERRUDFJAVUDT) <UDT Charset could not be set>” is returned.

### Syntax

```
void setBaseCharset(int  charset)
                      throws SQLException
```

## setBaseDataType(TeradataType)

Sets the base data type information for the UDT or complex type. Valid values include those contained in the Java enum `TeradataType`.

The method throws `SQLException` if an error occurs while attempting to set the base data type. A database specific code “9744 (ERRUDFJAVUDT) <UDT base type could not be set>” is returned.

### Syntax

```
void setBaseDataType(TeradataType  baseDataType)
                      throws SQLException
```

## setBaseMaxLength(int)

Sets the maximum length for the base type of the UDT or complex type.

The method throws `SQLException` if an error occurs while attempting to set the base maximum length. A database specific code “9744 (ERRUDFJAVUDT) <UDT base max length could not be set>” is returned.

### Syntax

```
void setBaseMaxLength(int  maxLength)
                      throws SQLException
```

## setBaseNumFractionalDigits(short)

Sets the fractional digits value for the base type of the UDT or complex type. This value corresponds to the value  $n$  in a `DECIMAL( $m,n$ )` base type; otherwise it is set to zero.



The method throws `SQLException` if an error occurs while attempting to set the fractional digits value. A database specific code “9744 (ERRUDFJAVUDT) <UDT base fractional digits could not be set>” is returned.

### Syntax

```
void setBaseNumFractionalDigits(short  numFractionalDigits)
                                throws SQLException
```

### setBasePrecision(int)

Sets the precision for the base type of the UDT or complex type. This value corresponds to the precision for time or timestamp types; for example `TIME(4)` or `TIMESTAMP(6)`, otherwise it is set to zero.

The method throws `SQLException` if an error occurs while attempting to set the precision value. A database specific code “9744 (ERRUDFJAVUDT) <UDT base precision could not be set>” is returned.

### Syntax

```
void setBasePrecision(int  precision)
                        throws SQLException
```

### setBaseTotalIntervalDigits(short)

Sets the total interval digits for the base type of the UDT or complex type. This value corresponds to the value *m* in a `DECIMAL(m,n)` base type; otherwise it is set to zero.

The method throws `SQLException` if an error occurs while attempting to set the total interval digits value. A database specific code “9744 (ERRUDFJAVUDT) <UDT base total interval digits could not be set>” is returned.

### Syntax

```
void setBaseTotalIntervalDigits(short  totalIntervalDigits)
                                throws SQLException
```

### setJSONStorageFormat(short)

Sets the JSON storage format for a JSON column. Valid values match those used for the C structure `json_storage_en`. For information about this structure, see [Table Operator Data Structures](#).

The method throws `SQLException` if an error occurs while attempting to set the JSON storage format value. A database specific code “9744 (ERRUDFJAVUDT) <JSON Storage Format could not be set>” is returned.

### Syntax

```
void setJSONStorageFormat(short indicator)
                        throws SQLException
```

### setStructNumAttributes(int)

Sets the number of total attributes for a structured UDT. This method is not applicable to any other UDT or complex types.

The method throws `SQLException` if an error occurs while attempting to set the number of attributes of the structured UDT. A database specific code “9744 (ERRUDFJAVUDT) <Structured UDT number of attributes could not be set>” is returned.

### Syntax

```
void setStructNumAttributes(int structNumAttributes)
                        throws SQLException
```

## setTransform Methods

Information about the transform type of a UDT can be set using the following methods:

- `void setTransformCharset(int c)`
- `void setTransformFractionalDigits(int d)`
- `void setTransformIntervalRange(int d)`
- `void setTransformMaxLength(int len)`
- `void setTransformPrecision(int p)`
- `void setTransformTotalDigits(int d)`
- `void setTransformType(TeradataType type)`

These methods are similar to the `setBase` type methods but refer to the default transform type of the UDT.

### setUdtIndicator(short)

Sets the UDT indicator attribute for the UDT or CDT.

Valid values match those used for the `udt_indicator` attribute of the C structure `UDT_BaseInfo_t`. For information about this structure, see [Table Operator Data Structures](#).

This method throws `SQLException` if an error occurs while attempting to set the indicator value. A database specific code “9744 (ERRUDFJAVUDT) <UDT indicator could not be set>” is returned.

### Syntax

```
void setUdtIndicator(short indicator)
                    throws SQLException
```

## setUdtName(java.lang.String)

Sets the name for a UDT or complex type.

The method throws `SQLException` if an error occurs while attempting to set the `udtName`. A database specific code “9744 (ERRUDFJAVUDT) <UDT name could not be set>” is returned.

### Syntax

```
void setUdtName(java.lang.String udtName)
                throws SQLException
```

## com.teradata.fnc.SQLXML

XML type parameters to a Java routine are mapped to `java.sql.SQLXML`. The `java.sql.SQLXML` interface provides methods for bringing the data of an SQL XML value to the client as a `String`, a `Reader` or `Writer`, or as a `Stream`. The `java.sql.SQLXML` interface is implemented by the `com.teradata.fnc.SQLXML` class.

The following sections describe the methods implemented by this class.

---

### Note:

The `getSource` and `setResult` methods are not supported.

---

## free()

This method closes the `SQLXML` object and releases the resources that it held.

### Syntax

```
void free()
    throws SQLException
```

### Usage Notes

The SQL XML object becomes invalid and is neither readable nor writeable when this method is called.

After free is called, any attempt to invoke a method other than free will result in an SQLException being thrown. If free is called multiple times, the subsequent calls to free are treated as a no-op.

## Exceptions

Throws SQLException if there is an error when trying to free the XML value. A database specific code "9752 (ERRUDFJAVAXML) <Failed to free XML object>" is returned.

## getBinaryStream()

Retrieves the XML value designated by this SQLXML instance as a stream.

Returns a stream containing the XML data.

The bytes of the input stream are interpreted according to the XML 1.0 specification.

## Syntax

```
InputStream getBinaryStream()
              throws SQLException
```

## Usage Notes

The behavior of this method is the same as ResultSet.getBinaryStream() when the designated column of the ResultSet has a type java.sql.Types of SQLXML.

The SQL XML object becomes unreadable when this method is called and may also become not writable depending on the implementation.

## Exceptions

Throws SQLException if there is an error processing the XML value. An exception is thrown if the state is not readable. A database specific code "9752 (ERRUDFJAVAXML) <Error retrieving XML value in getBinaryStream>" is returned.

## getCharacterStream()

Retrieves the XML value designated by this SQLXML instance as a java.io.Reader object.

Returns a stream containing the XML data.

The format of this stream is defined by org.xml.sax.InputSource, where the characters in the stream represent the Unicode code points for XML according to the XML 1.0 specification.

Although an encoding declaration other than Unicode may be present, the encoding of the stream is Unicode.

**Syntax**

```
Reader getCharacterStream()
        throws SQLException
```

**Usage Notes**

The behavior of this method is the same as `ResultSet.getCharacterStream()` when the designated column of the `ResultSet` has a type `java.sql.Types` of `SQLXML`.

The SQL XML object becomes unreadable when this method is called and may also become not writable depending on the implementation.

**Exceptions**

Throws `SQLException` if there is an error processing the XML value. The `getCause()` method of the exception may provide a more detailed exception, such as if the stream does not contain valid characters.

An exception is thrown if the state is not readable. A database specific code “9752 (ERRUDFJAVAXML) <Error retrieving XML value in getCharacterStream>” is returned.

**getString()**

Returns a string representation of the XML value designated by this `SQLXML` instance.

The format of this `String` is defined by `org.xml.sax.InputSource`, where the characters in the stream represent the Unicode code points for XML according to the XML 1.0 specification.

Although an encoding declaration other than Unicode may be present, the encoding of the `String` is Unicode.

**Syntax**

```
String getString()
        throws SQLException
```

**Usage Notes**

The behavior of this method is the same as `ResultSet.getString()` when the designated column of the `ResultSet` has a type `java.sql.Types` of `SQLXML`.

The SQL XML object becomes unreadable when this method is called and may also become not writable depending on the implementation.

**Exceptions**

Throws `SQLException` if there is an error processing the XML value. The `getCause()` method of the exception may provide a more detailed exception, such as if the stream does not contain valid characters.

An exception is thrown if the state is not readable. A database specific code “9752 (ERRUDFJAVAXML) <Error retrieving XML value in getString>” is returned.

## getUDFOutputSQLXML()

Returns the com.teradata.fnc.SQLXML object that represents the return XML value of the Java UDF.

### Syntax

```
public static java.sql.SQLXML getUDFOutputSQLXML()
    throws SQLException
```

### Usage Notes

This method can only be called from a scalar or aggregate Java UDF. If called from an aggregate Java UDF, it may only be called in the Phase.AGR\_FINAL execution phase.

### Exceptions

Throws SQLException if a database access error occurs. A database specific error code “9752 (ERRUDFJAVAXML) <Failed to retrieve XML object>” is returned.

## getXSPInoutSQLXMLForNull(int)

Returns the java.sql.SQLXML object that represents the INOUT parameter of a Java external stored procedure when a Null object is passed in for the parameter.

### Syntax

```
public static java.sql.SQLXML getXSPInoutSQLXMLForNull(int param_index)
    throws Exception
```

### Syntax Elements

#### *param\_index*

The index of the particular INOUT parameter as it appears in the list of parameters for the Java external stored procedure, starting from 0.

### Exceptions

Throws Exception if any error occurs during the process which could be either IOException or SQLException depending on the root cause of the error.

## setBinaryStream()

Retrieves a stream that can be used to write the XML value that this SQLXML instance represents.

Returns a stream to which data can be written.

The stream begins at position 0. The bytes of the stream are interpreted according to the XML 1.0 specification.

## Syntax

```
OutputStream setBinaryStream()  
              throws SQLException
```

## Usage Notes

The behavior of this method is the same as `ResultSet.updateBinaryStream()` when the designated column of the `ResultSet` has a type `java.sql.Types` of `SQLXML`.

The SQL XML object becomes not writeable when this method is called and may also become unreadable depending on the implementation.

## Exceptions

Throws `SQLException` if there is an error processing the XML value. An exception is thrown if the state is not writable.

A database specific code "9752 (ERRUDFJAVAXML) <Error processing XML value in setBinaryStream>" is returned.

## setCharacterStream()

Retrieves a stream to be used to write the XML value that this `SQLXML` instance represents.

Returns a stream to which data can be written.

The format of this stream is defined by `org.xml.sax.InputSource`, where the characters in the stream represent the Unicode code points for XML according to the XML 1.0 specification.

Although an encoding declaration other than Unicode may be present, the encoding of the stream is Unicode.

## Syntax

```
Writer setCharacterStream()  
       throws SQLException
```

## Usage Notes

The behavior of this method is the same as `ResultSet.updateCharacterStream()` when the designated column of the `ResultSet` has a type `java.sql.Types` of `SQLXML`.

The SQL XML object becomes not writeable when this method is called and may also become unreadable depending on the implementation.

## Exceptions

Throws SQLException if there is an error processing the XML value. The getCause() method of the exception may provide a more detailed exception, such as if the stream does not contain valid characters.

An exception is thrown if the state is not writable. A database specific code "9752 (ERRUDFJAVAXML) < Error processing XML value in setCharacterStream>" is returned.

## setString(String)

Sets the XML value designated by this SQLXML instance to the given String representation.

## Syntax

```
void setString(String value)
           throws SQLException
```

## Syntax Elements

### *value*

The string representation of the XML value.

The format of this String is defined by org.xml.sax.InputSource, where the characters in the stream represent the Unicode code points for XML according to the XML 1.0 specification.

Although an encoding declaration other than Unicode may be present, the encoding of the String is Unicode.

## Usage Notes

The behavior of this method is the same as ResultSet.updateString() when the designated column of the ResultSet has a type java.sql.Types of SQLXML.

The SQL XML object becomes not writeable when this method is called and may also become unreadable depending on the implementation.

## Exceptions

Throws SQLException if there is an error processing the XML value. The getCause() method of the exception may provide a more detailed exception, such as if the stream does not contain valid characters.

An exception is thrown if the state is not writable. A database specific code "9752 (ERRUDFJAVAXML) <Error processing XML value in setString>" is returned.

## Example: Using the com.teradata.fnc.SQLXML Class For an IN and OUT XML Parameter

This example returns an XML document that is the duplicate of the input XML document.



```

REPLACE PROCEDURE dupXML(IN xml_in XML,
                        OUT xml_out XML)

LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME
'UDF_JAR:UserDefinedFunctions.dupXML(java.sql.SQLXML, java.sql.SQLXML)';

public static void dupXML(java.sql.SQLXML xml_in,
                        java.sql.SQLXML xml_out) throws SQLException
{
    ...
}

```

## com.teradata.fnc.ST\_Geometry

ST\_Geometry data types can be optionally mapped to the com.teradata.fnc.ST\_Geometry Java class.

Geospatial data can be retrieved in either the Well-Known Text (WKT) or Well-Known Binary (WKB) format as a CLOB (for WKT) or a BLOB (for WKB) from an input ST\_Geometry value, and either format can be written to a return ST\_Geometry via the same respective CLOB and BLOB types.

The SRID also can be retrieved from input geometries and set in result geometries.

The following sections describe the methods implemented by this class.

### getSRID()

Returns an integer representing the Spatial Reference System Identifier (SRID) of an ST\_Geometry value. The ST\_Geometry value must be an input parameter to a Java UDF or an IN or INOUT parameter of a Java external stored procedure.

#### Syntax

```
public int getSRID() throws SQLException
```

#### Exceptions

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve SRID value>.

## getUDFOutputGeometry()

Returns a `com.teradata.fnc.ST_Geometry` object that represents the return `ST_Geometry` of the Java UDF. This method can be called only from a scalar or aggregate Java UDF. If the method is called from an aggregate Java UDF, it may only be called in the `Phase.AGR_FINAL` execution phase.

### Syntax

```
public com.teradata.fnc.ST_Geometry getUDFOutputGeometry()
    throws SQLException
```

### Exceptions

Throws `SQLException` if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve `ST_Geometry` object>.

## getWKB()

Returns the Well-Known Binary (WKB) representation of an `ST_Geometry` value as a `java.io.InputStream` object. The `ST_Geometry` value must be an input parameter to a Java UDF or an IN or INOUT parameter of a Java external stored procedure.

### Syntax

```
public java.io.InputStream getWKB()
    throws java.sql.SQLException
```

### Exceptions

Throws `SQLException` if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve WKB value>.

## getWKBSize()

Returns integer representing the size in bytes of Well-Known Binary (WKB) representation of an `ST_Geometry` value. The `ST_Geometry` value must be an input parameter to a Java UDF or an IN or INOUT parameter of a Java external stored procedure.

### Syntax

```
public long getWKBSize()
    throws java.sql.SQLException
```

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve WKB size>.

**getWKT()**

Returns the Well-Known Text (WKT) representation of an ST\_Geometry value as a java.io.InputStream object in the Latin character set. The ST\_Geometry value must be an input parameter to a Java UDF or an IN or INOUT parameter of a Java external stored procedure.

**Syntax**

```
public java.io.InputStream getWKT()
    throws java.sql.SQLException
```

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve WKT value>.

**getWKTSIZE()**

Returns an integer representing the size in bytes of the Well-Known Text (WKT) representation of an ST\_Geometry value. The ST\_Geometry value must be an input parameter to a Java UDF or an IN or INOUT parameter of a Java external stored procedure.

**Syntax**

```
public long getWKTSIZE()
    throws java.sql.SQLException
```

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve WKT size>.

**getXSPInoutGeometryForNull()**

Returns a com.teradata.fnc.ST\_Geometry object representing the INOUT parameter of a Java external stored procedure when a Null object is passed in for it.

**Syntax**

```
public com.teradata.fnc.ST_Geometry
    getXSPInoutGeometryForNull(int index)
        throws SQLException
```

**Syntax Elements***index*

The index of the particular INOUT parameter as it appeared in the list of the Java external stored procedure parameters. Index values begin from zero.

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to retrieve ST\_Geometry object>.

**setSRID()**

Sets the Spatial Reference System Identifier (SRID) of an ST\_Geometry value. The ST\_Geometry value must be the return parameter of a Java UDF or an OUT or INOUT parameter of a Java external stored procedure.

**Syntax**

```
public void setSRID(int srid) throws SQLException
```

**Syntax Elements***srid*

the Spatial Reference System Identifier of spatial coordinate system to be set.

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to set SRID value>.

**setWKB()**

Returns a java.io.OutputStream object that can be used to write the Well-Known Binary (WKB) value for a return ST\_Geometry value. The ST\_Geometry object must be the return value of a Java UDF or an OUT or INOUT parameter of a Java external stored procedure.

**Syntax**

```
public java.io.OutputStream setWKB()
    throws java.sql.SQLException
```

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to set WKB value>.

**setWKT()**

Returns a java.io.OutputStream object that can be used to write the Well-Known Text (WKT) value for a return ST\_Geometry value. The ST\_Geometry object must be the return value of a Java UDF or an OUT or INOUT parameter of a Java external stored procedure.

**Syntax**

```
public java.io.OutputStream setWKT()
    throws java.sql.SQLException
```

**Exceptions**

Throws SQLException if a database access error occurs. Returns the following database specific error code: 9747 (ERRUDFJAVAGEOM)<Failed to set WKT value>.

**Example: Using the com.teradata.fnc.ST\_Geometry Class For an IN and OUT ST\_Geometry Parameter**

This example returns a geometry that is the duplicate of the input geometry.

```
REPLACE PROCEDURE dupGeometry(IN geom_in ST_Geometry,
                              OUT geom_out ST_Geometry)
    LANGUAGE JAVA
    NO SQL
    PARAMETER STYLE JAVA
    EXTERNAL NAME
        'UDF_JAR:UserDefinedFunctions.dupGeometry(com.teradata.fnc.ST_Geometry,
com.teradata.fnc.ST_Geometry)';

public static void dupGeometry(com.teradata.fnc.ST_Geometry geom_in,
com.teradata.fnc.ST_Geometry[] geom_out) throws SQLException
{
    java.io.InputStream inWKT;
```

```

java.io.OutputStream outWKT;
byte [] wkt;
int srid;
int numbytes;

// Get the WKT InputStream and SRID of the input geometry.
inWKT = geom_in.getWKT();
srid = geom_in.getSRID();

// Read the input WKT into a byte array.
numbytes = inWKT.read(wkt);

// Retrieve an OutputStream where we can write the WKT to the return geoemtry.
If(geom_out == null | geom_out.length == 0)
    throw new SQLException();
outWKT = geom_out[0].setWKT();

// Write the input WKT to the return geometry
outWKT.write(wkt);

// Set the SRID of the return ST_Geometry to the same value as the input
geometry
geom_out[0].setSRID(srid);
}

```

## com.teradata.fnc.Struct

Structured UDT and Period data types are mapped to the `java.sql.Struct` interface. The `java.sql.Struct` interface is implemented by the `com.teradata.fnc.Struct` class.

The following sections describe the methods implemented by this class.

### getAttributes()

Returns an array containing the ordered attribute values of the SQL structured type that this `Struct` object represents.

---

#### Note:

Teradata does not support custom mapping, so only standard mapping is used.

---

**Syntax**

```
public Object[] getAttributes()
                throws SQLException
```

**Exceptions**

Throws SQLException if a database access error occurs. A database specific code “9742 (ERRUDFJAVGETATTRFAIL) <Failed to retrieve attributes of the Struct type >” is returned.

**getAttributes(Map)**

Returns an array containing the ordered attribute values of the SQL structured type that this Struct object represents.

**Note:**

Teradata does not support custom mapping, so only standard mapping is used.

**Syntax**

```
public Object[] getAttributes(Map map)
                throws SQLException
```

**Syntax Elements*****map***

A mapping of SQL type names to Java classes.

**Exceptions**

Throws SQLException if a database access error occurs. A database specific code “9742 (ERRUDFJAVGETATTRFAIL) <Failed to retrieve attributes of the Struct type >” is returned.

**getSQLTypeName()**

Returns the fully qualified type name of the SQL structured type for which this Struct object is the generic representation.

**Syntax**

```
public String getSQLTypeName()
                throws SQLException
```

## Exceptions

Throws SQLException if a database access error occurs. A database specific error code "9741 (ERRUDFJAVINVSQJNM) <SQL Type name could not be retrieved>" is returned.

## getXSPInoutStructForNull(int)

Returns the java.sql.Struct object representing the INOUT parameter of a Java external stored procedure when a Null object is passed in for the parameter.

## Syntax

```
java.sql.Struct getXSPInoutStructForNull(int param_index)
        throws Exception
```

## Syntax Elements

### *param\_index*

The index of the particular INOUT parameter as it appears in the list of parameters for the Java external stored procedure, starting from 0.

## Exceptions

Throws Exception if any error occurs during the process which could be either IOException or SQLException depending on the root cause of the error.

## Example: Using the java.sql.Struct Interface for a Structured UDT Parameter

```
CREATE TYPE EMPLOYEE AS (empid int,
                        first_name varchar(20),
                        last_name varchar(20),
                        salary decimal(10,2))

REPLACE PROCEDURE getEmployeeDetails(IN E1 Employee)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.getEmployeeDetails';

public static void getEmployeeDetails(java.sql.Struct E1) throws SQLException
{
    String objname;
    Object[] elements;
```



```

objname = E1.getSQLTypeName();
System.out.println("Object Name:"+ objname);

elements = E1.getAttributes();
System.out.println("Employee Id: " + (Integer)elements[0]);
System.out.println ("Employee name:  " + (String) elements[1] +
                    " " + (String)elements[2] );
System.out.println("Salary: "+ (BigDecimal)elements[3]);
}

```

## com.teradata.fnc.TeradataType

The com.teradata.fnc.TeradataType enum provides data type information about a particular column being passed to a table operator. The get method returns the TeradataType enum value corresponding to the integer value provided as input.

### enum TeradataType

The following lists a sample of the values defined in this enum.

```

public enum TeradataType {
    UNDEF_DT(0),
    CHAR_DT(1),
    VARCHAR_DT(2),
    BYTE_DT(3),
    VARBYTE_DT(4),
    GRAPHIC_DT(5),
    VARGRAPHIC_DT(6),
    BYTEINT_DT(7),
    SMALLINT_DT(8),
    INTEGER_DT(9),
    REAL_DT(10),
    DECIMAL1_DT(11),

    ...

    DATE_DT(15),
    TIME_DT(16),
    TIMESTAMP_DT(17),
    INTERVAL_YEAR_DT(18),

    ...
}

```

```

    TIME_WTZ_DT(31),
    TIMESTAMP_WTZ_DT(32),
    BLOB_REFERENCE_DT(33),
    CLOB_REFERENCE_DT(34),
    UDT_DT(35),
    BIGINT_DT(36),
    DECIMAL16_DT(37),
    NUMBER_DT(38),
    PERIOD_DT(39),
    JSON_DT(40),
    DATASET_AVRO_DT(41),
    ST_GEOMETRY_DT(42),
    MBR_DT(43),
    MBB_DT(44),
    ARRAY_DT(45),
    XML_DT(46),
    DATASET_CSV_DT(47),

    ...
}

```

## get(int)

Returns the TeradataType enum entry corresponding to the integer value provided as input. For example, the method returns XML\_DT for an input value of 46.

### Syntax

```

TeradataType get(int type)
                throws IllegalArgumentException

```

### Exceptions

The method throws IllegalArgumentException if an unsupported column type is encountered. A database specific code is returned with message "Unsupported column type: <type\_name>".

## com.teradata.fnc.Tbl

Provides methods for Java table UDF processing.

## Syntax

```
public class com.teradata.fnc.Tbl extends java.lang.Object{
    public static final int TBL_MODE_VARY;
    public static final int TBL_MODE_CONST;
    public static final int TBL_NOOPTIONS;
    public static final int TBL_LASTROW;
    public static final int TBL_NEWROW;
    public static final int TBL_NEWROEOF;
    public static final int TBL_PRE_INIT;
    public static final int TBL_INIT;
    public static final int TBL_BUILD;
    public static final int TBL_BUILD_EOF;
    public static final int TBL_FINI;
    public static final int TBL_END;
    public static final int TBL_ABORT;
    public static final int UNDEF_DT;
    public static final int CHAR_DT;
    public static final int VARCHAR_DT;
    public static final int BYTE_DT;
    public static final int VARBYTE_DT;
    public static final int GRAPHIC_DT;
    public static final int VARGRAPHIC_DT;
    public static final int BYTEINT_DT;
    public static final int SMALLINT_DT;
    public static final int INTEGER_DT;
    public static final int REAL_DT;
    public static final int DECIMAL1_DT;
    public static final int DECIMAL2_DT;
    public static final int DECIMAL4_DT;
    public static final int DECIMAL8_DT;
    public static final int DATE_DT;
    public static final int TIME_DT;
    public static final int TIMESTAMP_DT;
    public static final int INTERVAL_YEAR_DT;
    public static final int INTERVAL_YTM_DT;
    public static final int INTERVAL_MONTH_DT;
    public static final int INTERVAL_DAY_DT;
    public static final int INTERVAL_DTH_DT;
    public static final int INTERVAL_DTM_DT;
    public static final int INTERVAL_DTS_DT;
    public static final int INTERVAL_HOUR_DT;
    public static final int INTERVAL_HTM_DT;
```

```

public static final int INTERVAL_HTS_DT;
public static final int INTERVAL_MINUTE_DT;
public static final int INTERVAL_MTS_DT;
public static final int INTERVAL_SECOND_DT;
public static final int TIME_WTZ_DT;
public static final int TIMESTAMP_WTZ_DT;
public static final int BLOB_REFERENCE_DT;
public static final int CLOB_REFERENCE_DT;
public static final int UDT_DT;
public static final int BIGINT_DT;
public static final int DECIMAL16_DT;
public static final int NUMBER_DT;
public static final int FNC_DATATYPESETSIZE;
public static final int ERRUDFJTBLGETLENWCOL;
public static final int ERRUDFJTBLGETPRECSNWCOL;
public static final int ERRUDFJTBLGETSCALEWCOL;
public int getMode() throws java.io.IOException;
public int getPhase(int[]) throws java.io.IOException;
public int getPhaseEx(int[], int) throws java.io.IOException;
public com.teradata.fnc.Tbl();
public void allocCtx(java.lang.Object)
    throws java.io.IOException, java.sql.SQLException;
public void allocCtx(int)
    throws java.io.IOException, java.sql.SQLException;
public boolean control() throws java.io.IOException;
public int[] getColDef()
    throws java.io.IOException, java.sql.SQLException;
public int getLength(int)
    throws java.io.IOException, java.sql.SQLException;
public int getPrecision(int)
    throws java.io.IOException, java.sql.SQLException;
public int getScale(int)
    throws java.io.IOException, java.sql.SQLException;
public boolean optOut() throws java.io.IOException;
public boolean abort() throws java.io.IOException;
public boolean firstParticipant()
    throws java.io.IOException, java.sql.SQLException;
public void setCtrlCtx(java.lang.Object)
    throws java.io.IOException, java.sql.SQLException;
public java.lang.Object getCtrlCtx()
    throws java.io.IOException, java.lang.ClassNotFoundException,
    java.sql.SQLException;
public void setCtxObject(java.lang.Object)
    throws java.io.IOException, java.sql.SQLException;

```

```

public void setCtxObject(byte[])
    throws java.io.IOException, java.sql.SQLException;
public java.lang.Object getCtxObject()
    throws java.io.IOException, java.lang.ClassNotFoundException,
    java.sql.SQLException;
public byte[] getCtxObject(byte[])
    throws java.io.IOException, java.lang.ClassNotFoundException,
    java.sql.SQLException;
}

```

## Tbl Constructor

Constructs an instance of Tbl. Only table UDFs can construct an instance of Tbl.

### getMode()

Returns a value that indicates whether the arguments to the table function are constant or vary.

Value	Meaning
Tbl.TBL_MODE_CONST	The table UDF arguments are constant. The SELECT statement invoked the table function with constant expression input arguments.
Tbl.TBL_MODE_VARY	The table UDF arguments vary and are based on the rows produced by the correlated table specification in the SELECT statement. The table UDF might only be called on specific AMP vprocs in this mode.

### getPhase(int[] *phase*)

This method returns the current mode of the table function and the current phase value in the first element of the *phase* argument.

The processing phases that getPhase() can return depend on the mode:

Mode	Meaning
Tbl.TBL_MODE_CONST	The table UDF arguments are constant.
Tbl.TBL_MODE_VARY	The table UDF arguments vary and are based on the rows produced by the correlated table specification in the SELECT statement. The table UDF might only be called on specific AMP vprocs in this mode.

### Syntax

```
getPhase(int[] phase)
```

## Syntax Elements

### *phase*

Mode	<i>phase</i>	Meaning
Tbl.TBL_ MODE_ VARY	Tbl.TBL_ PRE_INIT	The table UDF is being called for the first time for all the rows that it will be called for. The input arguments to the UDF contain the first set of data.  During this phase, the UDF has an opportunity to establish overall global context, but should not build any result row. The UDF continues to the TBL_INIT phase.
	Tbl.TBL_ INIT	The input arguments to the UDF contain the first set of data. During this phase, the UDF should not build any result row. The UDF continues to the TBL_BUILD phase.
	Tbl.TBL_ BUILD	The UDF should fill out the result arguments to build a row. The UDF remains in the TBL_BUILD phase until it throws an SQLException with the sqlstate set to "02000" to indicate no data, whereupon it continues to the TBL_FINI phase.
	Tbl.TBL_ FINI	If there is more variable input data, the UDF returns to the TBL_INIT phase. Otherwise, the UDF continues to the TBL_END phase.
	Tbl.TBL_ END	The table function is not called again after this phase.
	Tbl.TBL_ ABORT	The table UDF is being aborted. A UDF can be invoked at any time with this phase, which is only entered when a copy of the table UDF invokes Tbl.abort(). It is not entered when the function is aborted for an external reason, such as a user abort.
Tbl.TBL_ MODE_ CONST	Tbl.TBL_ PRE_INIT	The UDF may decide whether it should be the controlling copy of all table functions running on other AMP vprocs.  If the function wants to provide control context to all other copies of the table function, the function must call Tbl.control(). If the function does not want to be the controlling copy of the table function, or if the function is designed without the need for a controlling function, the function can simply return and do nothing during this phase.  All copies of the table function must complete this phase before any copy continues to the TBL_INIT phase.
	Tbl.TBL_ INIT	Any copy of the UDF that does not want to participate further must call Tbl.optOut(). After the function returns, it is not called again.  All copies of the table UDF must complete this phase before any copy continues to the TBL_BUILD phase.
	Tbl.TBL_ BUILD	The table UDF should fill out the result arguments to build a row.

Mode	phase	Meaning
		The function remains in the Tbl.TBL_BUILD phase until it throws an SQLException, setting the sqlstate to "02000" to indicate no data, whereupon it continues to the Tbl.TBL_END phase.
	Tbl.TBL_END	The table function is not called again after it returns from this phase. The controlling copy of the table function, if one exists, is called with this phase after all other copies of the table function have completed this phase, which allows the controlling function to do any final cleanup or notification to the external world.
	Tbl.TBL_ABORT	The table UDF is being aborted and should perform cleanup, if necessary. A function can be called at any time with this phase, which is only entered when one of copies of the table function calls Tbl.abort(). It is not entered when the function is aborted for an external reason, such as a user abort.

## getPhaseEx(int[] phase, int option)

An alternative to getPhase() that provides additional options for variable mode table UDFs. getPhaseEx() provides a way for table functions to know when they are being passed in the last logical, or qualified row on an AMP. getPhaseEx() also provides users with more control of table phase transitions. Users can reduce the number of phase transitions required during execution of a table function, thus reducing the number of UDF invocations and improving table function performance.

The processing phases that getPhaseEx() can return depend on the mode:

Value	Meaning
Tbl.TBL_MODE_CONST	The table UDF arguments are constant. Although getPhaseEx() can be used successfully by a table function that is passed in constant arguments, it is most useful to a table function that is passed in variable arguments.
Tbl.TBL_MODE_VARY	The table UDF arguments vary and are based on the rows produced by the correlated table specification in the SELECT statement. The table UDF might only be called on specific AMP vprocs in this mode.

## Syntax

```
getPhaseEx(int[] phase, int option)
```

## Syntax Elements

### *phase*

The processing phases that `getPhaseEx()` returns in the *phase* argument depend on the mode.

Mode	<i>phase</i>	Meaning
Tbl.TBL_MODE_VARY	Tbl.TBL_PRE_INIT	The table UDF is being called for the first time for all the rows that it will be called for. The input arguments to the UDF contain the first set of data. During this phase, the UDF has an opportunity to establish overall global context, but should not build a result row. The UDF continues to the TBL_INIT phase.
	Tbl.TBL_INIT	The input arguments to the UDF contain the first set of data. During this phase, the UDF should not build a result row. The UDF continues to the TBL_BUILD phase.
	Tbl.TBL_BUILD	If the TBL_NEWROW option is set, then call the UDF with a new row with a phase of TBL_BUILD. If the TBL_NEWROWEOF option and EOF are set, then call the UDF with a new row with a phase of TBL_BUILD. If the TBL_LASTROW option is set, the function remains in the TBL_BUILD phase until it is passed in the last set of data, where it continues to the TBL_BUILD_EOF phase. The UDF can process any data that it wants to return in the TBL_BUILD_EOF phase. The UDF remains in the TBL_BUILD phase until it throws an SQLException with the sqlstate set to "02000" to indicate no data, whereupon it continues to the TBL_FINI phase.
	Tbl.TBL_BUILD_EOF	The UDF should fill out the result arguments to build a row. The UDF remains in the TBL_BUILD_EOF phase until it throws an SQLException with the sqlstate set to "02000" to indicate no data, whereupon it continues to the TBL_END phase.
	Tbl.TBL_FINI	The UDF returns to the TBL_INIT phase with more variable input data.
	Tbl.TBL_END	The table function is not called again after this phase.
	Tbl.TBL_ABORT	The table UDF is being aborted. A UDF can be invoked at any time with this phase, which is only entered when a copy of the table UDF invokes <code>Tbl.abort()</code> . It is not entered when the function is aborted for an external reason, such as a user abort.
Tbl.TBL_MODE_CONST	Tbl.TBL_PRE_INIT	The UDF may decide whether it should be the controlling copy of all table functions running on other AMP vprocs. If the function wants to provide control context to all other copies of the table function, the function must call <code>Tbl.control()</code> .



Mode	phase	Meaning
		<p>If the function does not want to be the controlling copy of the table function, or if the function is designed without the need for a controlling function, the function can simply return and do nothing during this phase.</p> <p>All copies of the table function must complete this phase before any copy continues to the TBL_INIT phase.</p>
	Tbl.TBL_INIT	<p>Any copy of the UDF that does not want to participate further must call Tbl.optOut(). After the function returns, it is not called again.</p> <p>All copies of the table UDF must complete this phase before any copy continues to the TBL_BUILD phase.</p>
	Tbl.TBL_BUILD	<p>The table UDF should fill out the result arguments to build a row. The function remains in the Tbl.TBL_BUILD phase until it throws an SQLException, setting the sqlstate to "02000" to indicate no data, whereupon it continues to the Tbl.TBL_END phase.</p>
	Tbl.TBL_END	<p>The table function is not called again after it returns from this phase.</p> <p>The controlling copy of the table function, if one exists, is called with this phase after all other copies of the table function have completed this phase, which allows the controlling function to do any final cleanup or notification to the external world.</p>
	Tbl.TBL_ABORT	<p>The table UDF is being aborted and should perform cleanup, if necessary. A function can be called at any time with this phase, which is only entered when one of copies of the table function calls Tbl.abort(). It is not entered when the function is aborted for an external reason, such as a user abort.</p>

### option

The following values are valid for the *option* argument only for variable mode table UDFs. These options are ignored if you specify them in a constant mode table UDF.

Options	Value	Description and Usage
Tbl.TBL_NOOPTIONS	0	<p>Indicates that no options are specified.</p> <p>Use this option when you only want to retrieve the processing phase in which the function was called.</p>
Tbl.TBL_LASTROW	1	<p>Allows a function to determine when it is being passed the last input row on an AMP.</p> <p>Use this option if your table function processes a set of input rows before returning an output row. You must set the EOF indicator when using TBL_LASTROW to signal the end of the processing when the last row has been encountered. The function then moves from the TBL_BUILD phase to the TBL_BUILD_EOF phase where the row is built.</p>

Options	Value	Description and Usage
Tbl.TBL_NEWROW	2	If this option is set and the phase is TBL_BUILD, the function is called with a new row with a phase of TBL_BUILD. Use this option when you want to get a new row on each function invocation. If end of file, TBL_LASTROW or ProcessLastRow is true, then this option is ignored.
Tbl.TBL_NEWROW and Tbl.TBL_LASTROW	3	The behavior for both the TBL_NEWROW and TBL_LASTROW options are in effect.
Tbl.TBL_NEWROWEOF	4	If this option is set, and the phase is TBL_BUILD and EOF is set, then the function is called with a new row with a phase of TBL_BUILD. Use this option when you want to get a new row when EOF is signaled. If end of file, TBL_LASTROW or ProcessLastRow is true, then this option is ignored.
Tbl.TBL_NEWROWEOF and Tbl.TBL_LASTROW	5	The behavior for both the TBL_NEWROWEOF and TBL_LASTROW options are in effect.

If you specify Tbl.TBL\_LASTROW, the option remains in effect for the duration of the request. The TBL\_NEWROW and TBL\_NEWROWEOF options are set and reset based on the options specified during each call to `getPhaseEx()` where the option value is nonzero.

For example, calling `getPhaseEx()` with the option Tbl.TBL\_NOOPTIONS does not reset the TBL\_NEWROW or TBL\_NEWROWEOF behavior.

## **allocCtx(java.lang.Object *tblCtxObj*)**

This method initializes the table function context for the first time with the serialized size of *tblCtxObj*.

A UDF can use the table function context as a scratchpad to retain data between iterations of a local table function copy.

### **Syntax**

```
allocCtx(java.lang.Object tblCtxObj)
```

## Syntax Elements

### *tblCtxObj*

If *tblCtxObj* has object type fields, the UDF must initialize those fields to non-null values before calling `allocCtx()`. If *tblCtxObj* has array type fields, the UDF must initialize the array itself and each element of the array to a non-null value before calling `allocCtx()`.

The maximum size of *tblCtxObj* is 64 KB.

## `allocCtx(int size)`

This method allocates a table function context and allows a UDF to use a byte array as a scratchpad or context object to retain data between iterations in the table UDF. Using a byte array allows context objects to be retrieved faster providing for better performance.

## Syntax

```
allocCtx(int size)
```

## Syntax Elements

### *size*

Size of the byte array in bytes.

## `control()`

Designates a table function as the controlling copy of all other copies of the table function running on other AMP vprocs.

This method returns true if the call is successful and the copy of table function can take the control role of the table function. If the method returns false, the copy of the table function should not assume the role of the control copy.

## Usage Notes

Setting up a controlling copy of a table function is useful when there is a need to distribute certain external control data among various copies of the table function that run on different AMP vprocs, but the external control data is on a particular node that only one copy of the table function can access.

Only one copy of the table function can successfully call `Tbl.control()`.

After calling `Tbl.control()`, the controlling copy of a table function can distribute control data to other table function copies by calling `Tbl.setCtrlCtx()`.

## getColDef()

Returns the definitions of the result columns that must be returned by a table function with dynamic result row specification.

The getColDef() method returns an array of int, where the array elements specify the data type of the result columns that the table function must return.

This method is only valid for table functions with dynamic result row specification.

### Usage Notes

The elements of the returned array specify the data types of the result columns that the table function must return. Valid values are defined by the following constants:

```
Tbl.UNDEF_DT
Tbl.CHAR_DT
Tbl.VARCHAR_DT
Tbl.BYTE_DT
Tbl.VARBYTE_DT
Tbl.GRAPHIC_DT
Tbl.VARGRAPHIC_DT
Tbl.BYTEINT_DT
Tbl.SMALLINT_DT
Tbl.INTEGER_DT
Tbl.REAL_DT
Tbl.DECIMAL1_DT
Tbl.DECIMAL2_DT
Tbl.DECIMAL4_DT
Tbl.DECIMAL8_DT
Tbl.DATE_DT
Tbl.TIME_DT
Tbl.TIMESTAMP_DT
Tbl.INTERVAL_YEAR_DT
Tbl.INTERVAL_YTM_DT
Tbl.INTERVAL_MONTH_DT
Tbl.INTERVAL_DAY_DT
Tbl.INTERVAL_DTH_DT
Tbl.INTERVAL_DTM_DT
Tbl.INTERVAL_DTS_DT
Tbl.INTERVAL_HOUR_DT
Tbl.INTERVAL_HTM_DT
Tbl.INTERVAL_HTS_DT
Tbl.INTERVAL_MINUTE_DT
Tbl.INTERVAL_MTS_DT
```

```
Tbl.INTERVAL_SECOND_DT
Tbl.TIME_WTZ_DT
Tbl.TIMESTAMP_WTZ_DT
Tbl.BLOB_REFERENCE_DT
Tbl.CLOB_REFERENCE_DT
Tbl.UDT_DT
Tbl.BIGINT_DT
Tbl.DECIMAL16_DT
Tbl.NUMBER_DT
```

For CHAR, BYTE, CHARACTER CHARACTER SET GRAPHIC, VARCHAR CHARACTER SET GRAPHIC, VARCHAR, VARBYTE, or NUMBER data types, call `Tbl.getLength()` to get the length of the data type.

For TIME or TIMESTAMP data types, call `Tbl.getPrecision()` to get the precision of the data type.

For a DECIMAL or NUMBER data types, call `Tbl.getPrecision()` to get the precision and `Tbl.getScale()` to get the scale of the data type.

### getLength(int *index*)

If a table function with dynamic result row specification calls `Tbl.getColDef()`, and the value of an element in the resulting array corresponds to a CHAR, BYTE, CHARACTER CHARACTER SET GRAPHIC, VARCHAR CHARACTER SET GRAPHIC, VARCHAR, VARBYTE or NUMBER data type, the table function can use `getLength()` to get the length of the data type.

#### Syntax

```
getLength(int index)
```

#### Syntax Elements

***index***

Index of array element whose length is to be returned.

#### Exceptions

If the index argument specifies an array element that is not a CHAR, BYTE, CHARACTER CHARACTER SET GRAPHIC, VARCHAR CHARACTER SET GRAPHIC, VARCHAR, VARBYTE or NUMBER column, `getLength()` throws an `SQLException` to indicate that `Tbl.getLength(int)` is invalid for the type of column intended, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"TS000"	7848

## getPrecision(int *index*)

If a table function with dynamic result row specification calls `Tbl.getColDef()`, and the value of an element in the resulting array corresponds to a TIME, TIMESTAMP, DECIMAL or NUMBER data type, the table function can use `getPrecision()` to get the precision of the data type.

### Syntax

```
getPrecision(int index)
```

### Syntax Elements

#### *index*

Index of array element whose precision is to be returned.

### Exceptions

If the index argument specifies an array element that is not a TIME, TIMESTAMP, DECIMAL or NUMBER column, `getPrecision()` throws an `SQLException` to indicate that `Tbl.getPrecision(int)` is invalid for the type of column intended, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"TS000"	7849

## getScale(int *index*)

If a table function with dynamic result row specification calls `Tbl.getColDef()`, and the value of an element in the resulting array corresponds to a DECIMAL or NUMBER data type, the table function can use `getScale()` to get the scale of the DECIMAL or NUMBER type.

### Syntax

```
getScale(int index)
```

### Syntax Elements

#### *index*

Index of array element whose scale is to be returned.

## Exceptions

If the index argument specifies an array element that is not a DECIMAL or NUMBER column, `getScale()` throws an `SQLException` to indicate that `Tbl.getScale(int)` is invalid for the type of column intended, setting the `SQLException` fields as follows.

SQLState Field	vendorCode Field
"TS000"	7850

## optOut()

Called by a copy of a table function that does not want to participate in the process of returning rows.

This method returns true if the call is successful and returns false otherwise.

## abort()

Provides a way for a table function to gracefully abort a request when it encounters an error condition and cannot continue.

This method returns true if the abort was initiated by this copy of the table function and return false if it was initiated by another copy of the table function.

This method is valid during any phase or mode in which the table function was invoked.

## firstParticipant()

Provides a way to implement a table function that only needs one copy to participate and does not care which AMP the copy runs on.

This method returns true if it is the first time being called and returns false otherwise.

The copy of the table function in which a call to `Tbl.firstParticipant()` returns true is the copy of the function that participates in the current transaction and request.

## setCtrlCtx(java.lang.Object tblCtrlCtx)

Stores an object in the table function control context.

## Usage Notes

A controlling copy of a table function (one that successfully calls `Tbl.control()`) can use the control context as a scratchpad to propagate data to all other copies of the UDF running on all other AMP vprocs.

## getCtrlCtx()

Returns the object that was stored in the table function control context with a previous call to `Tbl.setCtrlCtx()`.

## setCtxObject(java.lang.Object *tblCtxObj*)

Stores an object in the table function context.

A UDF can use the table function context as a scratchpad to retain data between iterations of a local table function copy.

## setCtxObject(byte[] *ctxByteArray*)

Stores the input byte array as the context object.

A UDF can use the byte array as a scratchpad or context object to retain data between iterations in the table UDF.

## getCtxObject()

Retrieves the object that was stored in the table function context with a previous call to `Tbl.allocCtx()` or `Tbl.setCtxObject()`.

## getCtxObject(byte[] *ctxByteArray*)

Retrieves the byte array that was stored as the context object with a previous call to `Tbl.allocCtx(int)` or `Tbl.setCtxObject(byte[])`.

## com.teradata.fnc.value.TeradataTime

Maps to an SQL TIME WITH TIME ZONE type parameter in a Java table operator.

All of the methods for the `java.sql.Time` class are available to an instance of the `TeradataTime` class.

## Syntax

```

public class  com.teradata.fnc.value.TeradataTime  extends java.sql.Time {
    public  TeradataTime(long);
    public void  setTimeZone(java.util.TimeZone);
    public java.util.TimeZone  getTimeZone();
}

```



## TeradataTime(long *time*)

Constructs a TeradataTime object.

### Syntax

```
TeradataTime(long time)
```

### Syntax Elements

*time*

The time value used to construct the TeradataTime object.

## setTimeZone(TimeZone *tz*)

Sets the time zone value for the TeradataTime object.

## getTimeZone()

Returns the time zone value of the TeradataTime object.

## com.teradata.fnc.value.TeradataTimestamp

Maps to an SQL TIMESTAMP WITH TIME ZONE type parameter in a Java table operator.

All of the methods for the java.sql.Timestamp class are available to an instance of the TeradataTimestamp class.

### Syntax

```
public class com.teradata.fnc.value.TeradataTimestamp extends
java.sql.Timestamp {
    public TeradataTimestamp(long);
    public void setTimeZone(java.util.TimeZone);
    public java.util.TimeZone getTimeZone();
}
```

## TeradataTimestamp(long *time*)

Constructs a TeradataTimestamp object.

## Syntax

```
TeradataTimestamp(long time)
```

## Syntax Elements

*time*

The time value used to construct the TeradataTime object.

## setTimeZone(TimeZone *tz*)

Sets the time zone value for the TeradataTimestamp object.

## Syntax

```
setTimeZone(TimeZone tz)
```

## Syntax Elements

*tz*

The time zone value for the TeradataTimestamp object.

## getTimeZone()

Returns the time zone value of the TeradataTimestamp object.

## com.teradata.fnc.TraceObj

Helper class to DbsInfo.traceWrite().

## Syntax

```
public class com.teradata.fnc.TraceObj extends java.lang.Object{
    public static final int UNKNOWN_J;
    public static final int INTEGER_J;
    public static final int BYTEINT_J;
    public static final int SHORT_J;
    public static final int DOUBLE_J;
    public static final int STRING_J;
    public static final int SQLDATE_J;
    public static final int SQLTIME_J;
    public static final int SQLTIMESTAMP_J;
```

```

public static final int BYTE_J;
public static final int BIGDECIMAL_J;
public static final int LONG_J;
public static final int UNKNOWN_DBS;
public static final int INTEGER_DBS;
public static final int BYTEINT_DBS;
public static final int SMALLINT_DBS;
public static final int DOUBLE_DBS;
public static final int DATE_DBS;
public static final int TIME_DBS;
public static final int TIMEWZONE_DBS;
public static final int TIMESTAMP_DBS;
public static final int TIMESTAMPWZONE_DBS;
public static final int REAL_DBS;
public static final int FLOAT_DBS;
public static final int VARBYTE_DBS;
public static final int BYTE_DBS;
public static final int DECIMAL_DBS;
public static final int NUMERIC_DBS;
public static final int BIGINT_DBS;
public static final int NUMBER_DBS;
public static final int INTERVALYEAR_DBS;
public static final int INTERVALYEARMON_DBS;
public static final int INTERVALMON_DBS;
public static final int INTERVALDAY_DBS;
public static final int INTERVALDAYHOUR_DBS;
public static final int INTERVALDAYMIN_DBS;
public static final int INTERVALDAYSEC_DBS;
public static final int INTERVALHOUR_DBS;
public static final int INTERVALHOURMIN_DBS;
public static final int INTERVALHOURSEC_DBS;
public static final int INTERVALMIN_DBS;
public static final int INTERVALMINSEC_DBS;
public static final int INTERVALSEC_DBS;
public static final int LATIN_VARCHAR_DBS;
public static final int LATIN_CHAR_DBS;
public static final int LATIN_LONGVARCHAR_DBS;
public static final int LATIN_CHARVARYING_DBS;
public static final int UNICODE_VARCHAR_DBS;
public static final int UNICODE_CHAR_DBS;
public static final int UNICODE_LONGVARCHAR_DBS;
public static final int UNICODE_CHARVARYING_DBS;
public static final int KANJISJIS_VARCHAR_DBS;
public static final int KANJISJIS_CHAR_DBS;

```

```

public static final int KANJISJIS_LONGVARCHAR_DBS;
public static final int KANJISJIS_CHARVARYING_DBS;
public static final int KANJI_VARCHAR_DBS;
public static final int KANJI_CHAR_DBS;
public static final int KANJI_LONGVARCHAR_DBS;
public static final int KANJI_CHARVARYING_DBS;
public com.teradata.fnc.TraceObj(java.lang.Object, int)
    throws java.lang.Exception;
public java.lang.Object getTraceObjObject();
public int getTraceObjJavaTypeID();
public int getTraceObjTraceTypeID();
}

```

## TraceObj Constructor

Use this constructor to wrap a Java object containing a value that you want to write to a column in a temporary trace table using `DbsInfo.traceWrite()`.

The first argument to this constructor is the Java object to wrap. The second argument is a static constant defined by the `TraceObj` class that identifies the SQL data type of the target column in the trace table. Use the following table to determine the type of Java object that you need to pass as the first argument and the static constant that you need to pass as the second argument.

IF the SQL data type of the trace table column is ...	THEN the first argument to this constructor is a ...	AND the second argument to this constructor is ...
INTEGER	<code>java.lang.Integer</code>	<code>TraceObj.INTEGER_DBS</code>
BYTEINT	<code>java.lang.Byte</code>	<code>TraceObj.BYTEINT_DBS</code>
SMALLINT	<code>java.lang.Short</code>	<code>TraceObj.SMALLINT_DBS</code>
BIGINT	<code>java.lang.Long</code>	<code>TraceObj.BIGINT_DBS</code>
DOUBLE	<code>java.lang.Double</code>	<code>TraceObj.DOUBLE_DBS</code>
DATE	<code>java.sql.Date</code>	<code>TraceObj.DATE_DBS</code>
TIME	<code>java.sql.Time</code>	<code>TraceObj.TIME_DBS</code>
TIME WITH TIME ZONE	<code>java.sql.Time</code>	<code>TraceObj.TIMEWZONE_DBS</code>
TIMESTAMP	<code>java.sql.Timestamp</code>	<code>TraceObj.TIMESTAMP_DBS</code>
TIMESTAMP WITH TIME ZONE	<code>java.sql.Timestamp</code>	<code>TraceObj.TIMESTAMPWZONE_DBS</code>
REAL	<code>java.lang.Double</code>	<code>TraceObj.REAL_DBS</code>
FLOAT	<code>java.lang.Double</code>	<code>TraceObj.FLOAT_DBS</code>
VARBYTE	<code>byte[]</code>	<code>TraceObj.VARBYTE_DBS</code>

IF the SQL data type of the trace table column is ...	THEN the first argument to this constructor is a ...	AND the second argument to this constructor is ...
BYTE	byte[]	TraceObj.BYTE_DBS
DECIMAL (n, m)	java.math.BigDecimal	TraceObj.DECIMAL_DBS
NUMERIC (n, m)	java.math.BigDecimal	TraceObj.NUMERIC_DBS
NUMBER(n, m)	java.math.BigDecimal	TraceObj.NUMBER_DBS
INTERVAL YEAR	java.lang.String	TraceObj.INTERVALYEAR_DBS
INTERVAL YEAR TO MONTH	java.lang.String	TraceObj.INTERVALYEARMON_DBS
INTERVAL MONTH	java.lang.String	TraceObj.INTERVALMON_DBS
INTERVAL DAY	java.lang.String	TraceObj.INTERVALDAY_DBS
INTERVAL DAY TO HOUR	java.lang.String	TraceObj.INTERVALDAYHOUR_DBS
INTERVAL DAY TO MINUTE	java.lang.String	TraceObj.INTERVALDAYMIN_DBS
INTERVAL DAY TO SECOND	java.lang.String	TraceObj.INTERVALDAYSEC_DBS
INTERVAL HOUR	java.lang.String	TraceObj.INTERVALHOUR_DBS
INTERVAL HOUR TO MINUTE	java.lang.String	TraceObj.INTERVALHOURMIN_DBS
INTERVAL HOUR TO SECOND	java.lang.String	TraceObj.INTERVALHOURSEC_DBS
INTERVAL MINUTE	java.lang.String	TraceObj.INTERVALMIN_DBS
CHAR CHARACTER SET LATIN	java.lang.String	TraceObj.LATIN_CHAR_DBS
VARCHAR CHARACTER SET LATIN	java.lang.String	TraceObj.LATIN_VARCHAR_DBS
CHAR VARYING CHARACTER SET LATIN	java.lang.String	TraceObj.LATIN_CHARVARYING_DBS
LONG VARCHAR CHARACTER SET LATIN	java.lang.String	TraceObj.LATIN_LONGVARCHAR_DBS
CHAR CHARACTER SET UNICODE	java.lang.String	TraceObj.UNICODE_CHAR_DBS
VARCHAR CHARACTER SET UNICODE	java.lang.String	TraceObj.UNICODE_VARCHAR_DBS
CHAR VARYING CHARACTER SET UNICODE	java.lang.String	TraceObj.UNICODE_CHARVARYING_DBS

IF the SQL data type of the trace table column is ...	THEN the first argument to this constructor is a ...	AND the second argument to this constructor is ...
LONG VARCHAR CHARACTER SET UNICODE	java.lang.String	TraceObj.UNICODE_ LONGVARCHAR_DBS

## getTraceObjObject()

Returns the object wrapped by the TraceObj object.

## getTraceObjectJavaTypeID()

Returns an integer value that the TraceObject uses to identify the type of object wrapped by the TraceObj object.

IF the object wrapped by the TraceObj object is a ...	THEN getTraceObjectJavaTypeID() returns ...
java.lang.Integer	TraceObj.INTEGER_J
java.lang.Byte	TraceObj.BYTEINT_J
java.lang.Short	TraceObj.SHORT_J
java.lang.Double	TraceObj.DOUBLE_J
java.sql.Date	TraceObj.SQLDATE_J
java.sql.Time	TraceObj.SQLTIME_J
java.sql.Timestamp	TraceObj.SQLTIMESTAMP_J
java.lang.Long	TraceObj.LONG_J
byte[]	TraceObj.BYTE_J
java.math.BigDecimal	TraceObj.BIGDECIMAL_J
java.lang.String	TraceObj.STRING_J

## getTraceObjectTraceTypeID()

Returns an integer value that identifies the SQL data type of the target column in the trace table. This value is the same as the value of the second argument passed in to the constructor.

## Related Information

FOR more information on ...	SEE ...
debugging a Java UDF or external stored procedure using trace tables	<a href="#">Debugging Using Trace Tables.</a>
CREATE GLOBAL TEMPORARY TRACE TABLE	<i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

# UDF Code Examples

This section provides C and Java code examples of scalar, aggregate, and table user-defined functions. You can also download UDFs from [Teradata Downloads](#).

## C Scalar Function

This example shows the C code for a simple scalar function that finds a specified pattern in a string. The function allows a maximum length of 500 characters for the string and the test pattern. The result is a CHAR value of 'T' for TRUE and 'F' for FALSE.

## SQL Definition

```
CREATE FUNCTION Find_Text( Searched_String VARCHAR(500),
                          Pattern VARCHAR(500) )
    RETURNS CHAR
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!pattern!dbs_functions/pattern.c!F!Find_Text';
```

## Example Query

The following query uses the Find\_Text function to find a specific text description in documents with a date not less than a certain age.

```
USING (age DATE, Look_For VARCHAR(500))
SELECT Doc_number, Text
FROM Documents
WHERE (:age <= Doc_Copyright)
AND
(Find_Text(Text,:Look_For) = 'T');
```

## C Function Definition

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
/* return a pointer to the SQLSTATE string */
```



```

void *Find_Text(VARCHAR_LATIN *searched_string,
                VARCHAR_LATIN *pattern,
                CHARACTER_LATIN *result,
                char  sqlstate[6])
{
    int pattern_found = 0;
    if (strlen((char *)pattern) == 0)
    {
        /* send a warning because probably a mistake */
        *result = 'T';
        strcpy(sqlstate, "01H01"); /* see SQLSTATE table */
        return;
    }
    /* do the pattern match -- code not shown */
    ...
    if (pattern_found)
        *result = 'T';
    else
        *result = 'F';
    /* SQLSTATE is initialized to zero so normal return does */
    /* not need to set it */
    return;
}

```

## C Scalar Function Using LOBs

This example implements a scalar function that takes three arguments, a BLOB, a VARBYTE, and another BLOB, and concatenates them in that order.

### SQL Definition

```

CREATE FUNCTION CONCAT3(A BLOB AS LOCATOR,
                        B VARBYTE(64000),
                        C BLOB AS LOCATOR)
    RETURNS BLOB AS LOCATOR
    LANGUAGE C
    NO SQL
    PARAMETER STYLE TD_GENERAL
    EXTERNAL NAME 'CS!udfsamp!td_udf/udfsamp.c';

```

## C Function Definition

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#define buffer_size 100000
const char trunc_err[6] = "22001";
static int append_lob(LOB_RESULT_LOCATOR, LOB_LOCATOR);
void concat3( LOB_LOCATOR      *a,
              VARBYTE          *b,
              LOB_LOCATOR      *c,
              LOB_RESULT_LOCATOR *result,
              char              sqlstate[6] )
{
    FNC_LobLength_t actlen;
    /* Append the first argument. Note that a truncation error could
       occur only if the result exceeds the maximum LOB size, so the
       first append should never get that error.
    */
    if (append_lob(*result, *a) != 0)
    {
        strcpy(sqlstate, trunc_err);
        return;
    }
    /* Append the second argument */
    if (FNC_LobAppend(*result, b->bytes, b->length, &actlen) != 0)
    {
        strcpy(sqlstate, trunc_err);
        return;
    }
    /* Append the third argument */
    if (append_lob(*result, *c) != 0)
    {
        strcpy(sqlstate, trunc_err);
        return;
    }
}

int append_lob( LOB_RESULT_LOCATOR dest,
               LOB_LOCATOR source )
{
    BYTE buffer[buffer_size];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;

```

```

int trunc_err = 0;
FNC_LobOpen(source, &id, 0, 0);
while( FNC_LobRead(id, buffer, buffer_size, &actlen) == 0 &&
      !trunc_err )
    trunc_err = FNC_LobAppend(dest, buffer, actlen, &actlen);
FNC_LobClose(id);
return trunc_err;
}

```

## C Scalar Function Using TD\_ANYTYPE Parameters

This example shows the C code for a simple scalar function that uses TD\_ANYTYPE input and result parameters. This function returns the ASCII numeric value of the first character of the input string. It accepts input strings with the following data types:

- CHARACTER
- VARCHAR
- CLOB
- UDT with a CHARACTER, VARCHAR, or CLOB attribute.
- ARRAY with an element type of CHARACTER or VARCHAR.

The function accepts the character set associated with the input string. If no character set is explicitly specified, the default character set is used.

The return type supported by this function is BYTEINT, SMALLINT, or INTEGER. To specify the desired return type for the function, use the RETURNS *data type* or RETURNS STYLE *column expression* clause when invoking the function.

## SQL Definition

```

CREATE FUNCTION ascii( str TD_ANYTYPE )
    RETURNS TD_ANYTYPE
    LANGUAGE C
    NO SQL
    SPECIFIC ascii
    EXTERNAL NAME 'CS!ascii!UDFs/ascii.c'
    PARAMETER STYLE SQL;

```

## Example Query

The following query returns the numeric representation of the first character in the string 'hello' according to the ASCII character encoding scheme.

```

SELECT (ascii('hello') RETURNS INTEGER);

```

The output returned by the query is:

```
ascii('hello')
-----
          104
```

## C Function Definition

```
#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"
#define IsNull -1
#define IsNotNull 0
#define NUL '\0'
#define ASCII_ANYTYPES 2

/* This function returns the ASCII numeric value of the first character of the
input string.
*/

void ascii(void      * inputStr,
            void      * result,
            int        * inputStrIsNull,
            int        * resultIsNull,
            char        sqlstate[6],
            SQL_TEXT    extname[129],
            SQL_TEXT    specific_name[129],
            SQL_TEXT    error_message[257])
{
    int numAnytypeParams;
    int buffersize;
    int returnValue;

    /* Returns NULL given NULL */
    if (*inputStrIsNull == IsNull)
    {
        *resultIsNull = IsNull;
        return;
    }

    buffersize = ASCII_ANYTYPES * sizeof(anytype_param_info_t);
    anytype_param_info_t * inputInfo = FNC_malloc(buffersize);
    FNC_GetAnytypeParamInfo( buffersize, &numAnytypeParams, inputInfo);
```

```

if ( numAnytypeParams >= 2
    && inputInfo[0].paramIndex == 1
    && ( inputInfo[0].datatype == VARCHAR_DT
        || inputInfo[0].datatype == CHAR_DT)
    )
{
    /* If the input is latin. */
    if(inputInfo[0].charset == LATIN_CT)
    {
        VARCHAR_LATIN * latinInputStr = (VARCHAR_LATIN *) inputStr;
        /* If the string is empty */
        if (*latinInputStr == NUL)
        {
            *resultIsNull = IsNull;
        }
        else
        {
            *resultIsNull = IsNotNull;
            returnValue = (INTEGER)(latinInputStr[0]);
        }
    }
    /* If the input is Unicode. */
    else if(inputInfo[0].charset == UNICODE_CT)
    {
        VARCHAR_UNICODE * unicodeInputStr = (VARCHAR_UNICODE *) inputStr;
        /* If the string is empty */
        if (*unicodeInputStr == NUL)
        {
            *resultIsNull = IsNull;
        }
        else
        {
            *resultIsNull = IsNotNull;
            returnValue = (INTEGER)(unicodeInputStr[0]);
        }
    }
    /* Deal with the output. */
    If (inputInfo[1].paramIndex == -1 && inputInfo[1].datatype == BYTEINT_DT)
    {
        * (BYTEINT *) result = (BYTEINT) returnValue;
    }
    else if ( inputInfo[1].paramIndex == -1
        && inputInfo[1].datatype == SMALLINT_DT)
    {

```

```

        * (SMALLINT *) result = (SMALLINT) returnValue;
    }
    else if ( inputInfo[1].paramIndex == -1
        && inputInfo[1].datatype == INTEGER_DT)
    {
        * (INTEGER *) result = returnValue;
    }
    else /* Error */
    {
        strcpy(sqlstate, "22023");
        strcpy((char *) error_message, "Invalid result type.");
    }
}
else /* Error */
{
    strcpy(sqlstate, "22023");
    strcpy((char *) error_message, "Invalid input type.");
}
/* Free memory */
FNC_free(inputInfo);
}

```

## C Scalar Functions for UDT Functionality

This example shows the C code for scalar functions that implement the following functionality for a structured UDT:

- transform functionality that allows exporting the UDT from the server
- transform functionality that allows importing the UDT to the server
- ordering functionality that allows two UDTs to be compared

## SQL Definition

Here is the SQL definition of a `color_t` distinct UDT and a `circle_t` structured UDT.

```

CREATE TYPE color_t
  AS VARCHAR(30)
  FINAL;

CREATE TYPE circle_t
  AS (x INTEGER, y INTEGER, radius INTEGER, color color_t)
  NOT FINAL;

```

The following CREATE FUNCTION statements install the functions:

```

CREATE FUNCTION circle_t_ToSql( p1 VARCHAR(80) )
  RETURNS circle_t
  NO SQL
  PARAMETER STYLE TD_GENERAL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!c_tosql!udfsrc/c_tosql.c!F!circle_t_ToSql';

CREATE FUNCTION circle_t_FromSql( p1 circle_t )
  RETURNS VARCHAR(80)
  NO SQL
  PARAMETER STYLE TD_GENERAL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!c_fromsql!udfsrc/c_fromsql.c!F!circle_t_FromSql';

CREATE FUNCTION circle_t_Ordering( p1 circle_t )
  RETURNS FLOAT
  SPECIFIC circle_t_Ordering
  NO SQL
  PARAMETER STYLE SQL
  DETERMINISTIC
  LANGUAGE C
  EXTERNAL NAME 'CS!c_order!udfsrc/c_order.c!F!circle_t_Ordering';

```

The following statement registers the *circle\_t\_ToSql* and *circle\_t\_FromSql* functions as transform routines for the *circle\_t* UDT:

```

CREATE TRANSFORM FOR circle_t
  circle_t_IO (TO SQL WITH SPECIFIC FUNCTION circle_t_ToSql,
    FROM SQL WITH SPECIFIC FUNCTION circle_t_FromSql);

```

The following statement registers the *circle\_t\_Ordering* function as an ordering routine for the *circle\_t* UDT:

```

CREATE ORDERING FOR circle_t
  ORDER FULL BY MAP WITH SPECIFIC FUNCTION circle_t_Ordering;

```

## Example Query

Consider this table that has a *circle\_t* column:

```

CREATE circle_table(pkey INTEGER, circle circle_t);

```

This query requires that a transform exists for the `circle_t` UDT that transforms to the client representation (in this case `VARCHAR(80)`):

```
SELECT * FROM circle_table;
```

This query requires that an ordering exists for the `circle_t` UDT:

```
SELECT *
FROM circle_table
WHERE circle < NEW circle_t().x(128).y(128).radius(1).color('RED');
```

## C Function Definition

The following example implements ordering functionality that allows Vantage to order two `circle_t` UDTs by their area. The function gets the radius and computes the area, returning the result as a `FLOAT` data type. If the circle is null or if the x attribute, y attribute, or radius attribute of the circle is null, the function returns a null result.

```
/* File: c_order.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_Ordering( UDT_HANDLE *circleUdt,
                       FLOAT      *result,
                       int         *indicator_circle,
                       int         *indicator_result,
                       char        sqlstate[6],
                       SQL_TEXT    extname[129],
                       SQL_TEXT    specific_name[129],
                       SQL_TEXT    error_message[257])
{
    INTEGER x, y, r;
    int nullIndicator;
    int length;

    /* If circle is null, return null. */
    if (*indicator_circle == -1) {
        *indicator_result = -1;
        return;
    }
}
```



```

    /* Verify that the x attribute is not null. */
    FNC_GetStructuredAttribute(*circleUdt, "x", &x, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == -1) {
        *indicator_result = -1;
        return;
    }

    /* Verify that the y attribute is not null. */
    FNC_GetStructuredAttribute(*circleUdt, "y", &y, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == -1) {
        *indicator_result = -1;
        return;
    }

    /* Get the radius attribute. */
    FNC_GetStructuredAttribute(*circleUdt, "radius", &r, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == -1) {
        *indicator_result = -1;
        return;
    }

    *result = 3.14 * r * r;
}

```

The following example implements transform functionality that transforms a VARCHAR(80) from the client to a *circle\_t* type on the server. The character string contains x, y, radius and color attributes separated by a colon (:) delimiter:

x:y:r:color

The absence of a value indicates a null attribute. For example, the format for a null radius is:

x:y::color

```

/* File: c_tosql.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_ToSql( VARCHAR_LATIN *p1,
                    UDT_HANDLE    *resultCircleUdt,

```

```

char          sqlstate[6])
{
    char buf[81];
    char *x_str, *y_str, *r_str, *color;
    char *p;
    INTEGER x, y, r;
    int nullIndicator;
    int len;

    /* Make a copy of the input string since we want to modify it. */
    strcpy(buf, (char *)p1);

    /* Retrieve and set the x coordinate field */
    x_str = buf;
    p = strchr(x_str, ':');
    *p = '\\0';
    if (x_str[0] == '\\0') {
        nullIndicator = -1;
    }
    else {
        nullIndicator = 0;
        x = atoi(x_str);
    }
    p++;
    FNC_SetStructuredAttribute(*resultCircleUdt, "x", &x,
nullIndicator, sizeof_INTEGER);

    /* Retrieve and set the y coordinate field */
    y_str = p;
    p = strchr(y_str, ':');
    *p = '\\0';
    if (y_str[0] == '\\0') {
        nullIndicator = -1;
    }
    else {
        nullIndicator = 0;
        y = atoi(y_str);
    }
    p++;
    FNC_SetStructuredAttribute(*resultCircleUdt, "y", &y,
nullIndicator, sizeof_INTEGER);

    /* Retrieve and set the radius field */
    r_str = p;

```

```

    p = strchr(r_str, ':');
    *p = '\0';
    if (r_str[0] == '\0') {
        nullIndicator = -1;
    }
    else {
        nullIndicator = 0;
        r = atoi(r_str);
    }
    p++;
    FNC_SetStructuredAttribute(*resultCircleUdt, "radius", &r,
nullIndicator, sizeof_INTEGER);

    /* Retrieve and set the color field */
    color = p;
    if (color[0] == '\0') {
        nullIndicator = -1;
    }
    else {
        nullIndicator = 0;
    }
    len = strlen(color);
    FNC_SetStructuredAttribute(*resultCircleUdt, "color", (VARCHAR_LATIN
*)color, nullIndicator, sizeof_VARCHAR_LATIN(len));
}

```

The following example implements transform functionality that allows Vantage to export a *circle\_t* type as a VARCHAR(80):

```

/* File: c_fromsql.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_FromSql( UDT_HANDLE    *circleUdt,
                      VARCHAR_LATIN *result,
                      char           sqlstate[6])
{
    INTEGER x, y, r;
    char y_str[11], r_str[11];
    VARCHAR_LATIN color[81];
    int nullIndicator;
    int length;

```

```

    /* Get each of the attributes and write them to the result string. */
    FNC_GetStructuredAttribute(*circleUdt, "x", &x, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == 0)
        sprintf((char *)result, "%d:", x);
    else
        strcpy((char *)result, ":");

    FNC_GetStructuredAttribute(*circleUdt, "y", &y, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == 0) {
        sprintf(y_str, "%d", y);
        strcat((char *)result, y_str);
    }
    strcat((char *)result, ":");

    FNC_GetStructuredAttribute(*circleUdt, "radius", &r, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == 0) {
        sprintf(r_str, "%d", r);
        strcat((char *)result, r_str);
    }
    strcat((char *)result, ":");

    FNC_GetStructuredAttribute(*circleUdt, "color", color,
sizeof_VARCHAR_LATIN_WITH_NULL(80), &nullIndicator, &length);
    if (nullIndicator == 0)
        strcat((char *)result, (char *)color);
}

```

## C Scalar Function Using Dynamic UDTs

This example shows how to write a UDF called `add_int` that handles a dynamic UDT input argument. The `add_int` UDF can handle a dynamic UDT that consists of any combination of the following attribute types:

The `add_int` UDF sums up the values of all of the attributes of the dynamic UDT (where attributes that are not numeric are expected to represent integers as strings) and returns the result.

- BYTEINT
- SMALLINT
- INTEGER
- CHAR
- VARCHAR

- CLOB

## SQL Definition

Create the function definition with a parameter that has a VARIANT\_TYPE data type.

```
CREATE FUNCTION add_int (p1 VARIANT_TYPE )
  RETURNS INTEGER
  NO SQL
  PARAMETER STYLE SQL
  RETURNS NULL ON NULL INPUT
  DETERMINISTIC LANGUAGE C
  EXTERNAL NAME 'CS!add_int!add_int.c!F!add_int';
```

## Example Query

Call the add\_int UDF, using the NEW VARIANT\_TYPE expression to create the dynamic UDT argument.

```
SELECT udf_scalar_mp_struc(NEW VARIANT_TYPE(1 AS x, '1' AS y,
  CAST('1' AS CLOB(8)) AS z));
```

## C Function Definition

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void add_int (UDT_HANDLE *dynUDT,
              INTEGER      *result,
              int           *indicator_udt,
              int           *indicator_result,
              char          sqlstate[6],
              SQL_TEXT      extname[129],
              SQL_TEXT      specific_name[129],
              SQL_TEXT      error_message[257])
{
  int Indicator=0;
  int length = 0;
  int sumValue=0;
  int AttributeIntValue = 0;
  char AttributeVCBuf[41];
  int AttributeVCInt=0;
  char AttributeLobBuf[41];
```

```

int AttributeLobInt=0;
LOB_LOCATOR AttributeLob;
FNC_LobLength_t StartLobLength;
FNC_LobLength_t MaxLobLength;
FNC_LobLength_t ActualLobLength;
LOB_CONTEXT_ID LobCtx;
UDT_HANDLE *UDT;
int Attribute_count=0;
int i;
attribute_info_t dyn_attrinfo;

strcpy((char *)sqlstate, "00000");
memset(&(error_message[0]), 0, sizeof(error_message));

/* This function takes a single Dynamic UDT parameter which is */
/* permitted to have an arbitrary number of expressions passed */
/* into it. Each expression is required to be one of:          */
/* - BYTEINT, SHORTINT, INTEGER value.                         */
/* - CHAR() string representing an integer value.             */
/* - VARCHAR() string representing an integer value.          */
/* - CLOB() string representing an integer value.              */

/* Determine number of expressions passed into */
/* NEW VARIANT_TYPE Dynamic UDT constructor */
FNC_GetStructuredAttributeCount(*dynUDT, &Attribute_count);

for(i=0; i<Attribute_count; i++)
{
    /* Get data type information about the attribute being processed */
    FNC_GetStructuredAttributeInfo(*dynUDT, i,
        sizeof(attribute_info_t), &dyn_attrinfo);

    /* Process the value based upon the data type */
    switch(dyn_attrinfo.data_type)
    {
        case BYTEINT_DT:
        case SMALLINT_DT:
        case INTEGER_DT:
            AttributeIntValue = 0;
            Indicator =0;
            /* Acquire the attribute value */
            FNC_GetStructuredAttribute(*dynUDT,
                &(dyn_attrinfo.attribute_name[0]), &AttributeIntValue,
                sizeof(INTEGER), &Indicator, &length);

```

```

        if (Indicator != 0)                // If NULL
            AttributeIntValue = 0;
        /* Process the value */
        sumValue = sumValue + AttributeIntValue;
        break;
case VARCHAR_DT:
case CHAR_DT:
    memset(&(AttributeVCBuf[0]), 0, sizeof(AttributeVCBuf));
    AttributeVCInt = 0;
    Indicator = 0;
    FNC_GetStructuredAttribute(*dynUDT,
        &(dyn_attrinfo.attribute_name[0]), &(AttributeVCBuf[0]),
        sizeof(AttributeVCBuf), &Indicator, &length);
    if (Indicator != 0)                // If NULL
        AttributeVCInt = 0;
    else
        AttributeVCInt = atoi(&(AttributeVCBuf[0]));
    /* Process the value */
    sumValue = sumValue + AttributeVCInt;
    break;
case CLOB_REFERENCE_DT:
    memset(&(AttributeLobBuf[0]), 0, sizeof(AttributeLobBuf));
    AttributeLobInt = 0;
    Indicator = 0;
    FNC_GetStructuredInputLobAttribute(*dynUDT,
        &(dyn_attrinfo.attribute_name[0]),
        &Indicator, &AttributeLob );
    if (Indicator != 0)                // If NULL
        AttributeLobInt = 0;
    else
    {
        MaxLobLength = 41;
        StartLobLength = 0;
        FNC_LobOpen(AttributeLob, &LobCtx, StartLobLength,
            MaxLobLength);
        FNC_LobRead(LobCtx, &(AttributeLobBuf[0]), MaxLobLength,
            &ActualLobLength);
        FNC_LobClose(LobCtx);
        AttributeLobInt = atoi(&(AttributeLobBuf[0]));
    }

    sumValue = sumValue + AttributeLobInt;
    break;
default:

```

```

        /* Invalid data type */
        strcpy((char *)sqlstate, "00023");
        sprintf(error_message,
            "\nBad data type passed in to Dynamic UDT");
        return;
        break;
    } /* end of the switch statement */
} /* end boundary for the for-attribute-processing-loop */
*indicator_result = 0;
*result = sumValue;
return;
}

```

## C Scalar Function for Compressing CLOB Data

This example shows a compression algorithm which removes all extra bytes of Unicode text and returns an error if the extra byte is not 0.

### NOTICE

You must thoroughly test the UDFs you develop for algorithmic compression. If the compression or decompression algorithm fails, compressed data may not be recoverable, or may be corrupted.

## SQL Definition

```

CREATE FUNCTION SYSUDTLIB.clobcompress
    (a CLOB AS LOCATOR CHARACTER SET UNICODE)
    RETURNS BLOB AS LOCATOR
    SPECIFIC clobcompress
    LANGUAGE C NO SQL
    FOR COMPRESS
    PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
    DETERMINISTIC
    EXTERNAL NAME 'cs!clobcompress!clobcompress.c';

```

## Example Table Definition

The following table definition specifies that the clobcompress and clobdecompress UDFs be used to compress and decompress the CLOB data in the gd\_data column.

```

CREATE TABLE GlobalData
    (gd_ID INTEGER,

```



```
gd_data CLOB CHARACTER SET UNICODE
  COMPRESS USING SYSUDTLIB.clobcompress
  DECOMPRESS USING SYSUDTLIB.clobdecompress);
```

## C Function Definition

```
#define SQL_TEXT Unicode_Text
#include "sqltypes_td.h"
#include <string.h>
#define buffer_size 64000

void clobcompress(LOB_LOCATOR*      sourceLobLoc,
                  LOB_RESULT_LOCATOR* resultLobLoc,
                  char               sqlstate[6] )

{
  BYTE inputBuffer[buffer_size];
  BYTE outputBuffer[buffer_size];
  LOB_CONTEXT_ID id;
  FNC_LobLength_t readlen, writelen;
  int trunc_err = 0;
  readlen = 0;
  writelen = 0;

  FNC_LobOpen(*sourceLobLoc, &id, 0, 0);

  while( FNC_LobRead(id, inputBuffer, buffer_size, &readlen) == 0 && !trunc_err )

  {
    /* Perform compression on the buffer (remove extra bytes)*/
    BYTE *inputPtr = inputBuffer;
    BYTE *outputPtr = outputBuffer;
    int i=0;
    for (i=0; i<(readlen/2); i++){
      memcpy(outputPtr,inputPtr,sizeof(BYTE));
      inputPtr += sizeof(BYTE);
      if (*inputPtr != 0)
      {
        strcpy(sqlstate,"38000",5);
        return;
      }
      inputPtr += sizeof(BYTE);
    }
  }
```

```

        outputPtr += sizeof(BYTE);
    }
    trunc_err = FNC_LobAppend(*resultLobLoc, outputBuffer, (readlen/2), &writelen);

    /* check trunc_err and properly report an error (performed in the same way
as for
    a standard UDF) */
}
FNC_LobClose(id);
}

```

## C Scalar Function for Decompressing CLOB Data

This example shows a decompression algorithm which reads text and adds the extra byte necessary to transform the text back to Unicode.

### NOTICE

You must thoroughly test the UDFs you develop for algorithmic compression. If the compression or decompression algorithm fails, compressed data may not be recoverable, or may be corrupted.

## SQL Definition

```

CREATE FUNCTION SYSUDTLIB.clobdecompress (a BLOB AS LOCATOR)
  RETURNS CLOB AS LOCATOR CHARACTER SET UNICODE
  SPECIFIC clobdecompress
  LANGUAGE C NO SQL
  FOR DECOMPRESS
  PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
  DETERMINISTIC
  EXTERNAL NAME 'cs!clobdecompress!clobdecompress.c';

```

## Example Table Definition

The following table definition specifies that the clobcompress and clobdecompress UDFs be used to compress and decompress the CLOB data in the gd\_data column.

```

CREATE TABLE GlobalData
  (gd_ID INTEGER,
   gd_data CLOB CHARACTER SET UNICODE
   COMPRESS USING SYSUDTLIB.clobcompress
   DECOMPRESS USING SYSUDTLIB.clobdecompress);

```

## C Function Definition

```

#define SQL_TEXT Unicode_Text
#include "sqltypes_td.h"
#include <string.h>
#define buffer_size 32000

void clobdecompress(LOB_LOCATOR*      sourceLobLoc,
                   LOB_RESULT_LOCATOR* resultLobLoc,
                   char                sqlstate[6] )
{
    BYTE inputBuffer[buffer_size];
    BYTE outputBuffer[buffer_size*2];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen;
    int trunc_err = 0;
    readlen=0;
    writelen=0;

    FNC_LobOpen(*sourceLobLoc, &id, 0, 0);
    while( FNC_LobRead(id, inputBuffer, buffer_size, &readlen) == 0 && !trunc_err )
    {
        int i=0;
        BYTE *inputPtr = inputBuffer;
        BYTE *outputPtr = outputBuffer;
        memset(outputBuffer,0,(buffer_size*2));
        for (i=0; i<readlen; i++)
        {
            memcpy(outputPtr,inputPtr,sizeof(BYTE));
            inputPtr += sizeof(BYTE);
            outputPtr += sizeof(BYTE)*2;
        }
        trunc_err = FNC_LobAppend(*resultLobLoc, outputBuffer, readlen*2, &writelen);

        /* check trunc_err and properly report an error (performed in the same way
as for
    a standard UDF) */
    }
    FNC_LobClose(id);
}

```

## C Scalar Function for Compressing Distinct UDT LOB Types

This example shows a compression algorithm which removes all extra bytes of Unicode text and returns an error if the extra byte is not 0.

### NOTICE

You must thoroughly test the UDFs you develop for algorithmic compression. If the compression or decompression algorithm fails, compressed data may not be recoverable, or may be corrupted.

### SQL Definition

```
CREATE FUNCTION SYSUDTLIB.dcllobcompress (a DCLOB)
  RETURNS BLOB AS LOCATOR
  SPECIFIC dcllobcompress
  LANGUAGE C NO SQL
  FOR COMPRESS
  PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
  DETERMINISTIC
  EXTERNAL NAME 'cs!dcllobcompress!dcllobcompress.c';
```

### Example Table Definition

The following table definition specifies that the dcllobcompress and dcllobdecompress UDFs be used to compress and decompress the data in the gd\_data column.

```
CREATE TABLE GlobalData
  (gd_ID INTEGER,
   gd_data DCLOB
   COMPRESS USING SYSUDTLIB.dcllobcompress
   DECOMPRESS USING SYSUDTLIB.dcllobdecompress);
```

### C Function Definition

```
#define SQL_TEXT Unicode_Text
#include "sqltypes_td.h"
#include <string.h>
#define buffer_size 64000

void dcllobcompress(UDT_HANDLE*      inputUDT,
                   LOB_RESULT_LOCATOR* resultLobLoc,
```

```

                                char                sqlstate[6] )
{
    LOB_LOCATOR sourceLobLoc;
    BYTE  inputBuffer[buffer_size];
    BYTE  outputBuffer[buffer_size];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen;
    int trunc_err = 0;
    readlen = 0;
    writelen = 0;

    FNC_GetDistinctInputLob(*inputUDT,&sourceLobLoc);
    FNC_LobOpen(sourceLobLoc, &id, 0, 0);

    while( FNC_LobRead(id, inputBuffer, buffer_size, &readlen) == 0 && !trunc_err )
    {
        /* Perform compression on the buffer (remove extra bytes)*/
        BYTE *inputPtr = inputBuffer;
        BYTE *outputPtr = outputBuffer;
        int i=0;
        for (i=0; i<(readlen/2); i++){
            memcpy(outputPtr,inputPtr,sizeof(BYTE));
            inputPtr += sizeof(BYTE);
            if (*inputPtr != 0)
            {
                strcpy(sqlstate,"38000",5);
                return;
            }
            inputPtr += sizeof(BYTE);

            outputPtr += sizeof(BYTE);
        }
        trunc_err = FNC_LobAppend(*resultLobLoc, outputBuffer, (readlen/2), &writelen);

        /* check trunc_err and properly report an error (performed in the same way
as for
    a standard UDF) */
    }
    FNC_LobClose(id);
}

```

## C Scalar Function for Decompressing Distinct UDT LOB Types

This example shows a decompression algorithm which reads text and adds the extra byte necessary to transform the text back to Unicode.

### NOTICE

You must thoroughly test the UDFs you develop for algorithmic compression. If the compression or decompression algorithm fails, compressed data may not be recoverable, or may be corrupted.

## SQL Definition

```
CREATE FUNCTION SYSUDTLIB.dclobdecompress (a BLOB AS LOCATOR)
  RETURNS DCLOB
  SPECIFIC dclobdecompress
  LANGUAGE C NO SQL
  FOR DECOMPRESS
  PARAMETER STYLE TD_GENERAL RETURNS NULL ON NULL INPUT
  DETERMINISTIC
  EXTERNAL NAME 'cs!dclobdecompress!dclobdecompress.c';
```

## Example Table Definition

The following table definition specifies that the dclobcompress and dclobdecompress UDFs be used to compress and decompress the data in the gd\_data column.

```
CREATE TABLE GlobalData
  (gd_ID INTEGER,
   gd_data DCLOB
   COMPRESS USING SYSUDTLIB.dclobcompress
   DECOMPRESS USING SYSUDTLIB.dclobdecompress);
```

## C Function Definition

```
#define SQL_TEXT Unicode_Text
#include "sqltypes_td.h"
#include <string.h>
#define buffer_size 32000

void dclobdecompress(LOB_LOCATOR*      sourceLobLoc,
                    UDT_HANDLE*         resultUDT,
```

```

char sqlstate[6] )
{
    LOB_RESULT_LOCATOR resultLobLoc;
    BYTE inputBuffer[buffer_size];
    BYTE outputBuffer[buffer_size*2];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t readlen, writelen;
    int trunc_err = 0;
    readlen=0;
    writelen=0;

    FNC_GetDistinctResultLob(*resultUDT,&resultLobLoc);
    FNC_LobOpen(*sourceLobLoc, &id, 0, 0);
    while( FNC_LobRead(id, inputBuffer, buffer_size, &readlen) == 0 && !trunc_err )
    {
        int i=0;
        BYTE *inputPtr = inputBuffer;
        BYTE *outputPtr = outputBuffer;
        memset(outputBuffer,0,(buffer_size*2));
        for (i=0; i<readlen; i++)
        {
            memcpy(outputPtr,inputPtr,sizeof(BYTE));
            inputPtr += sizeof(BYTE);
            outputPtr += sizeof(BYTE)*2;
        }
        trunc_err = FNC_LobAppend(resultLobLoc, outputBuffer, readlen*2, &writelen);

        /* check trunc_err and properly report an error (performed in the same way
as for
a standard UDF) */
    }
    FNC_LobClose(id);
}

```

## C Scalar Functions for Row Level Security

This example shows a typical use of row level security to protect sensitive data. The implementation is based on a sensitivity label assigned to each user of the system and to the rows of the permanent tables to be protected. A security policy is defined to:

- Enforce the sensitivity label that is assigned to new and/or updated rows.
- Control access to the rows of a data set based on the sensitivity label of the user versus the sensitivity label of the rows.
- Determine the markings that must be applied to sensitive data exported from the system.

## Security Constraint Object Definition

This example defines two constraint objects which together comprise a sensitivity label.

```
CREATE CONSTRAINT Classification_Level SMALLINT, NOT NULL,
  VALUES (TopSecret:4, Secret:3, Confidential:2, Unclassified:1),
  INSERT SYSLIB.InsertLevel,
  UPDATE SYSLIB.UpdateLevel,
  DELETE SYSLIB.DeleteLevel,
  SELECT SYSLIB.ReadLevel;

CREATE CONSTRAINT Classification_Category BYTE(8),
/* leave room for 64 categories, nulls are allowed */
  VALUES (NATO:1, UnitedStates:2, Canada:3, UnitedKingdom:4,
    France:5, Norway:6, Russia:7),
  INSERT SYSLIB.InsertCategory,
  UPDATE SYSLIB.UpdateCategory,
  DELETE SYSLIB.DeleteCategory,
  SELECT SYSLIB.ReadCategory;
```

## SQL Function Definition

The function to insert a row takes the session level value as input and returns the level to be assigned to the new row.

```
CREATE FUNCTION SYSLIB.InsertLevel( current_session SMALLINT )
  RETURNS SMALLINT
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'CS!insertlevel!udfsrc/insertlevel.c';
```

The function to update a row takes the session level value and the target row level value as input. It returns the level to be assigned to the updated row.

```
CREATE FUNCTION SYSLIB.UpdateLevel( current_session SMALLINT,
                                     input_row        SMALLINT )
  RETURNS SMALLINT
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'CS!updatelevel!udfsrc/updatelevel.c';
```



The function to delete a row takes the target row level value as input and returns 'Y' to indicate that the DELETE request is allowed or 'N' to indicate that the request is denied.

```
CREATE FUNCTION SYSLIB.DeleteLevel( input_row SMALLINT )
  RETURNS CHAR
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'CS!deletelevel!udfsrc/deletelevel.c';
```

The function to select a row takes the session level value and the target row level value as input. It returns 'Y' to indicate that the SELECT request is allowed or 'N' to indicate that the request is denied.

```
CREATE FUNCTION SYSLIB.ReadLevel( current_session SMALLINT,
                                input_row          SMALLINT )
  RETURNS CHAR
  LANGUAGE C
  NO SQL
  PARAMETER STYLE TD_GENERAL
  EXTERNAL NAME 'CS!readlevel!udfsrc/readlevel.c';
```

The function to insert a row takes the session category value as input and returns the category to be assigned to the new row.

```
CREATE FUNCTION SYSLIB.InsertCategory( current_session BYTE(8) )
  RETURNS BYTE(8)
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!insertcategory!udfsrc/insertcategory.c';
```

The function to update a row takes the session category value and the target row category value as input. It returns the category to be assigned to the updated row.

```
CREATE FUNCTION SYSLIB.UpdateCategory( current_session BYTE(8),
                                      input_row          BYTE(8) )
  RETURNS BYTE(8)
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!updatecategory!udfsrc/updatecategory.c';
```

The function to delete a row takes the target row category value as input and returns 'Y' to indicate that the DELETE request is allowed or 'N' to indicate that the request is denied.

```
CREATE FUNCTION SYSLIB.DeleteCategory( input_row BYTE(8) )
  RETURNS CHAR
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!deletecategory!udfsrc/deletecategory.c';
```

The function to select a row takes the session category value and the target row category value as input. It returns 'Y' to indicate that the SELECT request is allowed or 'N' to indicate that the request is denied.

```
CREATE FUNCTION SYSLIB.ReadCategory( current_session BYTE(8),
                                     input_row        BYTE(8) )
  RETURNS CHAR
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!readcategory!udfsrc/readcategory.c';
```

## C Function Definition

The following shows sample C code for the UDFs that will enforce the security policies. The policies used in the example are:

- For INSERT operations, the new row must contain the sensitivity label of the session.
- For UPDATE operations, the session sensitivity label must dominate the target row label or the update is not allowed. If the UPDATE is allowed, then the label in the updated row must contain the level of the session and a category which is a combination of the categories from the session and the current row.
- For DELETE operations, the row cannot be deleted unless the sensitivity level is unclassified and the sensitivity category is null.
- For SELECT operations, the session sensitivity label must dominate the target row label.

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

typedef unsigned char byte;

// INSERT Level UDF
void InsertLevel( short int  *sess_level,
                  short int  *new_row )
{
```

```

    // Level of new row equals that of the session
    // return level to DBS
    *new_row = *sess_level;
    return;
}

// UPDATE Level UDF

void UpdateLevel( short int  *sess_level,
                  short int  *curr_row,
                  short int  *upd_row )
{
    // Session's level must be equal to or greater than row's to allow
    // the update. If update is allowed, then the level of the updated
    // row is changed to that of the session.

    // can the session update the row?
    if (*sess_level >= *curr_row)
        // set level in updated row to session's level
        *upd_row = *sess_level;
    else
        // update is not allowed
        *upd_row = 0;
    return;
}

//DELETE Level UDF

void DeleteLevel( short int  *curr_row,
                  char        *result )
{
    // Row can only be deleted if it has a label of unclassified
    if (*curr_row == 1)
        // delete is allowed
        *result = 'T';
    else
        *result = 'F';
    return;
}

// SELECT Level UDF

void ReadLevel( short int  *sess_level,
                short int  *curr_row,

```

```

        char      *result )
{
    // Read allowed if session's level is equal to or greater than row's
    if (*sess_level >= *curr_row)
        // select is allowed
        *result = 'T';
    else
        //select is not allowed
        *result = 'F';
    return;
}

//INSERT Category UDF

void InsertCategory( byte sess_cat[],
                    byte new_row[],
                    int *indic1,
                    int *retindic)
{
    int i;

    // policy is that category for new row equals that for the session

    if (*indic1 == -1) // null session, then null new row
        *retindic = -1;
    else // non-null session, then new row equals session
    {
        *retindic = 0;
        for (i = 0; i < 8; i++)
            new_row[i] = sess_cat[i];
    }
    return;
}

// UPDATE Category UDF

void UpdateCategory( byte sess_cat[],
                    byte curr_row[],
                    byte new_row[],
                    int *indic1,
                    int *indic2,
                    int *retindic)
{

```

```

int i;

// Policy is that row can be updated if user's category is a
// superset of the row's category. If the update is allowed,
// then the category for the new row is a combination of the
// categories of the current row plus those of the session.

if (*indic1 == -1) // if session's category is null
{
    if (*indic2 == -1) // if row's category is null, update is allowed
    { // both are null
        *retindic = -1; // new row constraint is null
        return;
    }
    else
    { // session is null and row is not, so disallow update
        *retindic = 0;
        for (i = 0; i < 8; i++)
            new_row[i] = 0; // set return category to null for error
        return;
    }
}
// session's category is not null
*retindic = 0; // must return a category
if (*indic2 == -1) // if row's category is null, then update
                  // is allowed
{
    for (i = 0; i < 8; i++)
        new_row[i] = sess_cat[i]; //set return category to session
    return;
}
// session's category and input row's category are not null
for (i = 0; i < 8; i++) // does session dominate row?
    if ((sess_cat[i] & curr_row[i]) != curr_row[i])
        // no it does not, so update is not allowed
        {
            for (i = 0; i < 8; i++)
                new_row[i] = 0; // set return category to all zeroes
            //for error

            return;
        }
// update is allowed; set return category to a combination of
// session's and row's categories

```

```

    for (i = 0; i < 8; i++)
        new_row[i] = sess_cat[i] | curr_row[i]; //inclusive or
    return;
}

// DELETE Category UDF

void DeleteCategory( byte curr_row[],
                    char *result,
                    int *indic1,
                    int *retindic)

{
    // Policy is that row can be deleted only if its category is null
    *retindic = 0; // result is always not null
    if (*indic1 == -1) // if row's category is not null
        *result = 'T'; // delete is allowed
    else
        *result = 'F'; // delete is not allowed
    return;
}

// SELECT Category UDF

void ReadCategory( byte sess_cat[],
                  byte curr_row[],
                  char *result,
                  int *indic1,
                  int *indic2,
                  int *retindic)

{
    int i;

    // Policy is that row can be selected if user's category is a
    // superset of the row's category.
    *retindic = 0; // result is always not null
    *result = 'T';
    if ((*indic1 == *indic2) && (*indic1 == -1)) // both are null
        return; // allow select
    if (*indic2 == -1) // row is null and session isn't
        return; // allow select
    // both are not null
    *result = 'F';

```

```

    for (i = 0; i < 8; i++) // does session dominate row
        // does session dominate
        if ((curr_row[i] ^ sess_cat[i]) & curr_row[i])
            return; // no, so select is not allowed
    *result = 'T'; // yes, so allow select
    return;
}

```

## C Aggregate Function

This example shows the C code for a simple aggregate UDF that calculates the standard deviation, as follows:

$$s = \sqrt{\frac{\sum X^2}{N} - \left(\frac{\sum X}{N}\right)^2}$$

## SQL Definition

```

CREATE FUNCTION STD_DEV(x FLOAT)
    RETURNS FLOAT
    CLASS AGGREGATE
    LANGUAGE C
    NO SQL
    PARAMETER STYLE SQL
    EXTERNAL;

```

## Example Query

```

SELECT Product_ID, SUM(Hours), STD_DEV(Hours)
FROM Product_Life
WHERE Product_Class = 'Bulbs'
GROUP BY Product_ID;

```

## C Function Definition

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <math.h>
typedef struct agr_storage {
    FLOAT count;

```

```

        FLOAT    x_sq;
        FLOAT    x_sum;
    } AGR_Storage;

void STD_DEV( FNC_Phase      phase,
              FNC_Context_t *fctx,
              FLOAT          *x,
              FLOAT          *result,
              int            *x_i,
              int            *result_i,
              char           sqlstate[6],
              SQL_TEXT       fncname[129],
              SQL_TEXT       sfncname[129],
              SQL_TEXT       error_message[257] )
{
    /* pointers to intermediate storage areas */
    AGR_Storage *s1 = fctx->interim1;
    AGR_Storage *s2 = fctx->interim2;

    /* The standard deviation function described here is: */
    /*                                                     */
    /*  $s = \sqrt{\text{sum}(x^2)/N - (\text{sum}(x)/N)^2}$  */
    /*                                                     */
    /* sum( $x^2$ ) :> x_sq, N :> count, sum(x) :> x_sum */
    /*                                                     */

    /* switch to determine the aggregation phase */
    switch (phase)
    {
        /* This case selection is called once per group and */
        /* allocates and initializes all intermediate */
        /* aggregate storage */
        /* */
        case AGR_INIT:
            /* Get some storage for intermediate aggregate values. */
            /* FNC_DefMem returns zero if it cannot get the requested */
            /* memory. The amount of storage required is the size of */
            /* the structure used for intermediate aggregate
values */

            s1 = FNC_DefMem(sizeof(AGR_Storage));
            if (s1 == NULL)
            {
                /* could not get storage */
                strcpy(sqlstate, "U0001"); /* see SQLSTATE table */
            }
        }
    }
}

```



```

        return;
    }
    /* Initialize the intermediate aggregate values */
    s1->count = 0;
    s1->x_sq = 0;
    s1->x_sum = 0;

    /******
    /* Fall through to detail phase, because the      */
    /* AGR_INIT call passes in the first set of      */
    /* values for the group                          */
    /******
    /* This case selection is called once for each    */
    /* selected row to aggregate. x is the column the */
    /* std_dev is being calculated for.              */
    /* One copy will be running on each AMP          */

    case AGR_DETAIL:
    if (*x_i != -1)
    {
        s1->count++;
        s1->x_sq += *x * *x;
        s1->x_sum += *x;
    }
    break;
    /* This case selection combines the results of ALL */
    /* individual AMPs for each group                  */

    case AGR_COMBINE:
    s1->count += s2->count;
    s1->x_sq += s2->x_sq;
    s1->x_sum += s2->x_sum;
    break;
    /* This case selection returns the final standard */
    /* deviation. It is called once for each group.   */

    case AGR_FINAL:
    {
        FLOAT term2 = s1->x_sum/s1->count;
        FLOAT variance = s1->x_sq/s1->count - term2*term2;
        /* Adjust for deviations close to zero */
        if (fabs(variance) < 1.0e-14)
            variance = 0.0;
        *result = sqrt(variance);
    }

```

```

        break;
    }
    case AGR_NODATA:
        /* return null if no data */
        *result_i = -1;
        break;

    default:
        /* If it gets here there must be an error because this
        */
        /* function does not accept any other phase options */
        strcpy(sqlstate, "U0005");
    }
    return;
}

```

## C Aggregate Function Using LOBs

This example implements the MAX\_BLOB aggregate function and illustrates the use of persistent object references (LOB\_REF).

Like the standard MAX aggregate function, it returns the greatest BLOB value of the given set. The comparison rule is the same as for VARBYTE or BYTE type values; that is, trailing zeros are not significant.

The algorithm assumes that the objects to be compared are random-valued byte strings. A complete ordering of a pair of strings can be determined by examining the first unequal byte pair.

In the vast majority of cases, the first unequal byte pair can be found within the first PREFIX\_SIZE (500) bytes. By saving the prefix of the winner of each comparison, the need to reread any object is rare.

## SQL Definition

```

CREATE FUNCTION MAX_BLOB(X BLOB AS LOCATOR)
  RETURNS BLOB AS LOCATOR(1000)
  CLASS AGGREGATE
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  EXTERNAL NAME 'CS!Max_Blob!td_udf/max.c';

```

## C Function Definition

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>

```

```

#include <string.h>
#define BUFFER_SIZE 100000
#define PREFIX_SIZE 500
/* Aggregate intermediate record */
typedef struct
{
    FNC_LobLength_t length;
    int             initflag;
    LOB_REF         ref;
    BYTE            prefix[PREFIX_SIZE];
} intrec_t;
/*****
    load_rec() Initialize the intrec structure for a specified
                object;
*****/
static void
load_rec (intrec_t* rec, LOB_LOCATOR loc)
{
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actsize;
    rec->length = FNC_GetLobLength(loc);
    rec->initflag = 1;
    FNC_LobOpen(loc, &id, 0, PREFIX_SIZE);
    FNC_LobRead(id, rec->prefix, PREFIX_SIZE, &actsize);
    FNC_LobClose(id);
    /* if the object length is less than the size allotted for the
       prefix, pad the remaining bytes with zeros. */
    while (actsize < PREFIX_SIZE)
        rec->prefix[actsize++] = 0;
    FNC_LobLoc2Ref(loc, &rec->ref);
}
/*****
    findnz() Scan the open stream and return 1 if a nonzero
              byte is found, else return 0.
*****/
static int
findnz(LOB_CONTEXT_ID id)
{
    FNC_LobLength_t i, actsize;
    BYTE buffer[BUFFER_SIZE];
    while (FNC_LobRead(id, buffer, BUFFER_SIZE, &actsize) == 0)
    {
        for (i = 0; i < actsize; ++i)
            if (buffer[i] != 0)

```

```

        return 1;
    }
    return 0;
}
/*****
    greater()    Compare 2 byte strings and return 1 if a > b,
                -1 if a < b, else return 0.
*****/
static int
greater(BYTE* a, BYTE* b, int length)
{
    for (; length > 0; ++a, ++b, --length)
    {
        if ( *a > *b)
            return 1;
        else if (*a < *b)
            return -1;
    }
    return 0;
}
/*****
    rest_greater() Compare the rest of 2 open streams (a, b) and
                  return 1 if a > b, else return 0.
*****/
static int
rest_greater(LOB_CONTEXT_ID a, LOB_CONTEXT_ID b)
{
    FNC_LobLength_t i, actsizea, actsizeb;
    BYTE bufa[BUFFER_SIZE/2], bufb[BUFFER_SIZE/2];
    while (FNC_LobRead(a, bufa, sizeof(bufa), &actsizea) == 0)
    {
        if (FNC_LobRead(b, bufb, actsizea, &actsizeb) == 0)
        {
            int cmpres = greater(bufa, bufb, actsizeb);
            if (cmpres != 0)
                return cmpres;
            if (actsizea > actsizeb)
            {
                /* exhausted b before a */
                for (i = actsizeb; i < actsizea; ++i)
                    if (bufa[i] != 0)
                        return 1;
                return findnz(a);
            }
        }
    }
}

```

```

    }
    else
    {
        /* b exhausted; then a > b iff the rest of a contains
           any nonzero byte */
        return findnz(a);
    }
}
/* Reached the end of a without exhausting b; ergo a >= b */
return 0;
}
/*****
greater_obj()   Compare 2 objects (a, b) using cached prefix
                strings when possible, or by reading whole objects
                when necessary. Return 1 if a > b, else return 0.
*****/
static int
greater_obj(intrec_t* a, intrec_t* b)
{
    LOB_CONTEXT_ID streama, streamb;
    int cmpres = greater(a->prefix, b->prefix, PREFIX_SIZE);
    /* If the comparison was resolved in the prefix, then we
       are done */
    if (cmpres != 0)
        return cmpres;
    if (a->length < PREFIX_SIZE)
        return 0; /* must be a <= b */
    /* We will have to read the rest of object a */
    FNC_LobOpen(FNC_LobRef2Loc(&a->ref), &streama,
                PREFIX_SIZE, 0);
    if (b->length < PREFIX_SIZE)
    {
        /* a > b iff substr(a, PREFIX_SIZE+1) has any
           nonzero byte */
        cmpres = findnz(streama);
    }
    else
    {
        /* We will have to read the rest of object b also */
        FNC_LobOpen(FNC_LobRef2Loc(&b->ref), &streamb,
                    PREFIX_SIZE, 0);
        /* Compare the rest of the 2 objects */
        cmpres = rest_greater(streama, streamb);
        FNC_LobClose(streamb);
    }
}

```

```

    }
    FNC_LobClose(streama);
    return cmpres;
}

/*****
    append_lob()  Copy source to dest
*****/
static void
append_lob(LOB_RESULT_LOCATOR dest, LOB_LOCATOR source )
{
    BYTE buffer[BUFFER_SIZE];
    LOB_CONTEXT_ID id;
    FNC_LobLength_t actlen;
    int trunc_err = 0;
    FNC_LobOpen(source, &id, 0, 0);
    while( FNC_LobRead(id, buffer, BUFFER_SIZE, &actlen) == 0 &&
        !trunc_err)
        trunc_err = FNC_LobAppend(dest, buffer, actlen, &actlen);
    FNC_LobClose(id);
}

/*****
    Max_Blob()      Main function
*****/
void Max_Blob(FNC_Phase phase,
              FNC_Context_t* fctx,
              LOB_LOCATOR* x,
              LOB_RESULT_LOCATOR* result,
              int* x_i,
              int* result_i,
              char sqlstate[6])
{
    intrec_t detail_rec;
    intrec_t* s1 = (intrec_t*) fctx->interim1;
    intrec_t* s2 = (intrec_t*) fctx->interim2;
    switch (phase)
    {
    case AGR_INIT:
        if ((s1 = (intrec_t*)FNC_DefMem(sizeof(intrec_t))) == NULL)
        {
            /* could not get storage */
            strcpy(sqlstate, "U0001");
            return;
        }
        s1->initflag = 0;
    }
}

```

```

    /* Now fall through to detail phase logic */

case AGR_DETAIL:
    if (*x_i == -1)
        return;
    if (s1->initflag)
    {
        load_rec(&detail_rec, *x);
        if (greater_obj(&detail_rec, s1) == 1)
            *s1 = detail_rec;
    }
    else
    {
        s1->initflag = 1;
        load_rec(s1, *x);
    }
    break;
case AGR_COMBINE:
    if (greater_obj(s2, s1) == 1)
        *s1 = *s2;
    break;
case AGR_FINAL:
    if (s1->initflag)
        append_lob(*result, FNC_LobRef2Loc(&s1->ref));
    else
        *result_i = -1; /* set result to null */
    break;
case AGR_NODATA:
    *result_i = -1; /* set result to null */
    break;
default:
    strcpy(sqlstate, "U0005");
}
}

```

## C Aggregate Function Using TD\_ANYTYPE Parameters

This example shows the C code for a simple aggregate function that uses a TD\_ANYTYPE input parameter.

### SQL Definition

```

CREATE FUNCTION AggFncWAnytype (p1 TD_ANYTYPE)
    RETURNS INTEGER

```

```

CLASS AGGREGATE (8)
LANGUAGE C
NO SQL
EXTERNAL NAME
'CS!AggFncWAnytype!UDFS/AggFncWAnytype.c!F!AggFncWAnytype'
PARAMETER STYLE SQL;

```

## Example Query

Assume that there is a table ByteIntTable defined as follows:

```
SELECT * FROM ByteIntTable ORDER BY 1;
```

```
*** Query completed. 6 rows found. 3 columns returned.
```

```
*** Total elapsed time was 1 second.
```

RowNumber	ByteIntColumn	Remark
1	1	Nominal Value
2	-128	Lower Limit
3	127	Upper Limit
4	0	Zero Value
5	?	NULL Value
6	50	Character Literal

The following is a sample query that calls the AggFncWAnytype function.

```

SELECT (AggFncWAnytype (ByteIntColumn) RETURNS INT)
FROM ByteIntTable ORDER BY 1;

```

The output from the query is:

```

AggFncWAnytype(1)
-----
                10

```

## C Function Definition

```

/* function context area */
typedef struct arg_storage {
    int  MaxOfColumn ;
} AgrStorage ;

```



```

void AggFncWAnytype(
    FNC_Phase      phase,
    FNC_Context_t  *fContext,
    void           *input,
    INTEGER        *result,
    int            *inputIsNull,
    int            *resultIsNull,
    char           sqlstate[6],
    SQL_TEXT       extname[129],
    SQL_TEXT       specific_name[129],
    SQL_TEXT       error_message[257] )

{
    int            sumValue = 0;
    anytype_param_info_t paraminfo;
    int            numAnyType;

    AgrStorage *s1 = fContext->interim1 ;
    AgrStorage *s2 = fContext->interim2 ;

    switch (phase)
    {
        case AGR_INIT:
            /* Allocate and initialize the intermediate storage. */
            if ((s1 = FNC_DefMem(sizeof(AgrStorage))) == NULL) {
                strcpy((char *)sqlstate, "U09999") ;
                strcpy((char *)&(error_message[0]), "Memory allocation failure");
                return;
            }

            s1->MaxOfColumn = 0;
            strcpy((char *)sqlstate, "00000");
            memset(&(error_message[0]), 0, sizeof(error_message));

        case AGR_DETAIL:
            FNC_GetAnyTypeParamInfo( sizeof(anytype_param_info_t), &numAnyType, &paraminfo);

            /* Process the value based upon the data type. */
            switch(paraminfo.datatype)
            {
                case BYTEINT_DT:
                case SMALLINT_DT:
                case INTEGER_DT:

```

```

        /* Process the value */
        s1->MaxOfColumn = 10;
        break;
    case VARCHAR_DT:
    case CHAR_DT:
        s1->MaxOfColumn = 11;
        break;
    case CLOB_REFERENCE_DT:
        s1->MaxOfColumn = 12;
        break;
    default:
        /* Invalid string */
        strcpy((char *)sqlstate, "22023");
        sprintf((char*)error_message, "\n Invalid Data Type passed into UDF");
        break;
} /* end of the switch statement */
break;

case AGR_COMBINE:
    /* combine the intermediate results */
    if(s1->MaxOfColumn < s2->MaxOfColumn)
        s1->MaxOfColumn =s2->MaxOfColumn;
    break;

case AGR_FINAL:
    /* produce the final result */
    *result = s1->MaxOfColumn;
    break;

case AGR_NODATA:
    /* No data passed in ... so nothing to do */
    //Indicate that OUTPUT is being produced.
    *resultIsNull =IsNull;
    break;

default:
    strcpy (sqlstate, "U0005") ;
    sprintf((char*)error_message, "\n Unknown Aggregate Phase!");
    return;
}
}

```

## C Aggregate Function Using UDTs

This example shows how to write an aggregate UDF that takes a distinct UDT argument and returns a distinct UDT argument. The predefined type on which the distinct UDT is based is VARCHAR(20000) and the UDF performs a concatenation on the aggregation group.

### SQL Definition

Here is the SQL definition of a VARCHARUDT distinct UDT.

```
CREATE TYPE VARCHARUDT AS VARCHAR(20000) FINAL;
```

The table AggrDataTable defines a VARCHARUDT column.

```
CREATE TABLE AggrDataTable (aggrID int, aggrName varcharudt);
```

The following CREATE FUNCTION statement installs the aggregate function:

```
CREATE FUNCTION udf_agch002002udt (parameter_1 VARCHARUDT)
  RETURNS VARCHARUDT
  CLASS AGGREGATE (20000)
  LANGUAGE C
  NO SQL
  EXTERNAL NAME 'CS!udf_agch002002udt!udf_agch002002udt.c'
  PARAMETER STYLE SQL;
```

### Example Query

Consider the following data in the AggrDataTable table:

```
INSERT INTO AggrDataTable VALUES (1, 'george');
INSERT INTO AggrDataTable VALUES (1, 'carlos');
INSERT INTO AggrDataTable VALUES (2, 'annabelle');
INSERT INTO AggrDataTable VALUES (2, 'eva');
```

The following query performs the aggregation (concatenation) on the VARCHARUDT column:

```
SELECT aggrID, udf_agch002002udt(aggrName)
FROM AggrDataTable
GROUP BY aggrID;
```

## C Function Definition

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

/* Aggregate storage area */
typedef struct arg_storage {
    CHARACTER ConcatOfColumn[20000] ;
} AgrStorage ;

void udf_agch002002udt(FNC_Phase      phase,
                      FNC_Context_t *fContext,
                      UDT_HANDLE      *ColumnValueUdt,
                      UDT_HANDLE      *resultUdt,
                      int              *ColumnValueUdtIndicator,
                      int              *ResultUdtIndicator,
                      char             sqlstate[6],
                      SQL_TEXT        ExternalName[129],
                      SQL_TEXT        SpecificName[129],
                      SQL_TEXT        ErrorMsg[257])
{
    char ColumnValue[20000]; // Buffer to hold retrieved column value
    int length;             // Length of retrieved value
    char result[20000];
    AgrStorage *s1 = fContext->interim1 ; // Pointers to intermediate
    AgrStorage *s2 = fContext->interim2 ; // storage areas

    /* Switch to determine aggregation phase */
    switch (phase)
    {
        case AGR_INIT:
            /* Allocate and initialize the intermediate storage */
            if ((s1 = FNC_DefMem(sizeof(AgrStorage))) == NULL) {
                strcpy((char *)sqlstate, "U09999") ;
                strcpy((char *)ErrorMsg, "Memory allocation failure");
                return;
            }
            strcpy((char*)s1->ConcatOfColumn, "INIT");
            strcpy((char *)sqlstate, "00000");
            *ResultUdtIndicator = -1;
            /* Drop through to first row data */
        case AGR_DETAIL:

```

```

    if (*ColumnValueUdtIndicator != -1) { // if not NULL
        FNC_GetDistinctValue(*ColumnValueUdt, &ColumnValue,
                            sizeof(ColumnValue), &length);
        /* Concatenate values */
        strcat((char*)s1->ConcatOfColumn, (char *)ColumnValue);
    }
    *ResultUdtIndicator = -1;
    break ;
case AGR_COMBINE:
    /* Aggregate groups. */
    strcat((char*)s1->ConcatOfColumn, (char*)s2->ConcatOfColumn);
    *ResultUdtIndicator = -1;
    break ;
case AGR_FINAL:
    /* Produce the final result */
    strcat((char*)s1->ConcatOfColumn, "FINAL");
    strcpy((char *)result, (char*)s1->ConcatOfColumn);
    FNC_SetDistinctValue(*resultUdt, &result,
                        sizeof(s1->ConcatOfColumn));
    *ResultUdtIndicator = 0;
    break;
case AGR_NODATA:                                     // Set indicator to NULL
    *ResultUdtIndicator = -1 ;
    break ;
default:
    strcpy (sqlstate, "U0005") ;
}
return ;
}

```

## C Window Aggregate Function

This example shows the C code for a window aggregate UDF that performs the dense rank operation.

### SQL Definition

```

REPLACE FUNCTION dense_rank (x INTEGER)
  RETURNS INTEGER
  CLASS AGGREGATE (1000)
  LANGUAGE C
  NO SQL
  PARAMETER STYLE SQL
  DETERMINISTIC

```

```

CALLED ON NULL INPUT
EXTERNAL;

```

## Example Query

The `dense_rank` UDF evaluates dense rank over the set of values passed as arguments to the UDF. With dense ranking, items that compare equal receive the same ranking number, and the next item(s) receive the immediately following ranking number.

Consider the following table definition and inserted data:

```

CREATE MULTISET TABLE t
(id INTEGER,
 v  INTEGER);

INSERT INTO t VALUES (1,1);
INSERT INTO t VALUES (1,2);
INSERT INTO t VALUES (1,2);
INSERT INTO t VALUES (1,4);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,8);
INSERT INTO t VALUES (1,);

```

In the following query and result, note the difference in the rank and dense rank value for `v=4`. The dense rank value is 4 whereas the rank of 4 is 5.

```

SELECT v, dense_rank(v) OVER (PARTITION BY id ORDER BY v
    ROWS UNBOUNDED PRECEDING) as dr,
    rank() OVER (PARTITION BY id ORDER BY v) as r
FROM t ORDER BY dr;

```

The output from the `SELECT` statement is:

v	dr	r
-----	-----	-----
?	1	1
1	2	2
2	3	3
2	3	3
4	<b>4</b>	<b>5</b>
5	5	6
5	5	6

5	5	6
8	6	9

## C Function Definition

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

typedef struct agr_storage {
    int cr;    // current rank
    int pv;    // previous value
} AGR_Storage;

void dense_rank( FNC_Phase      phase,
                 FNC_Context_t *fctx,
                 INTEGER        *x,
                 INTEGER        *result,
                 int             *x_i,
                 int             *result_i,
                 char            sqlstate[6],
                 SQL_TEXT       fncname[129],
                 SQL_TEXT       sfncname[129],
                 SQL_TEXT       error_message[257] )
{
    /* pointer to intermediate storage area */
    AGR_Storage *s1 = fctx->interim1;

    /* switch to determine the aggregation phase */
    switch (phase)
    {
        case AGR_INIT:
            /* This UDF currently handles only the cumulative window */
            /* type for illustrative purposes. */
            if (fctx->window_size != -1)
            {
                strcpy(error_message, "Only cumulative window type
supported");

                strcpy(sqlstate, "U0001"); /* see SQLSTATE table */
                return;
            }
            if ( (s1=FNC_DefMem(sizeof(AGR_Storage))) == NULL)
            {
```

```

        strcpy(sqlstate, "U0002");
        return;
    }
    s1->cr = 1;
    s1->pv = *x;

    /******
    /* Fall through to the detail phase
    /******

    case AGR_DETAIL:
    if (*x != s1->pv)
    {
        s1->cr++;
        s1->pv = *x;
    }
    break;

    case AGR_FINAL:
        *result = s1->cr;
    break;

    /* Add this to generate an error for any undefined phases */
    case AGR_COMBINE:
    case AGR_MOVINGTRAIL:
    default:
        sprintf(error_message, "phase is %d", phase);
        strcpy(sqlstate, "U0005");
        return;
    }
}

```

## C Table Function

This example shows the C code for a table function. The example uses the following tables, data, and include file:

```

CREATE TABLE t (v INTEGER);

INSERT INTO t (1);
INSERT INTO t (2);
INSERT INTO t (3);
INSERT INTO t (4);

```



```

INSERT INTO t (5);
INSERT INTO t (6);
INSERT INTO t (7);
INSERT INTO t (8);
INSERT INTO t (9);
INSERT INTO t (10);

CREATE MULTISET GLOBAL TEMPORARY TRACE TABLE udftrace ,NO FALLBACK ,
    CHECKSUM = DEFAULT,
    NO LOG
    (
        Amp_number BYTE(2),
        Sequence INTEGER,
        Message VARCHAR(500)
    );

```

The phasetrace.h include file contains the following:

```
void trace (char *input_string);
```

## SQL Definition

```

REPLACE FUNCTION PHASE
    (in1 INTEGER,
     in2 VARCHAR(32) CHARACTER SET LATIN)
RETURNS TABLE
    (o1 INTEGER,
     o2 VARCHAR(500) CHARACTER SET LATIN)
SPECIFIC fnc_phase
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'CI!phasetrace!phasetrace.h!CS!fnc_phase!fnc_phase.c';

```

## Example Query

```

SET SESSION FUNCTION TRACE USING 'Play' FOR TABLE udftrace;

SELECT * FROM table (phase (t.v, 'mike')) AS d
; SELECT * FROM udftrace ORDER BY 1,2;

```

## C Function Definition

```
#define SQL_TEXT Latin_Text
#define SQL_STATE_LENGTH 6
#define ERROR_MESSAGE_LENGTH 257

#include <sqltypes_td.h>
#include "phasetrace.h"

void trace (char *input_string)
{
    void *argv[1];
    int length[1];
    argv[0] = input_string;
    length[0] = strlen(input_string);
    FNC_Trace_Write_DL(1, argv, length);    /* write to trace table. */
}

char tbuf[256];

typedef struct
{
    int cnt;
} local_ctx;

void fnc_phase(
    INTEGER *in1,
    VARCHAR_LATIN *in2,
    INTEGER *out1,    // first result attribute
    VARCHAR_LATIN *out2,
    int* i_i1,
    int* i_i2,
    int* i_o1,
    int* i_o2,
    char      sqlstate[SQL_STATE_LENGTH],
    SQL_TEXT  fncname[FNC_MAXNAMELEN],
    SQL_TEXT  sfncname[FNC_MAXNAMELEN],
    SQL_TEXT  errmsg[ERROR_MESSAGE_LENGTH])

{
    FNC_Phase  Phase;
    local_ctx *state_info;

    /* make sure the function is called in the supported context */
}
```

```

if (FNC_GetPhase(&Phase) != TBL_MODE_VARY)
{
    strcpy(sqlstate, "U0005");
    strcpy((char *) errmsg, "Table function being called in unsupported mode.");
    return;
}
switch(Phase)
{
    case TBL_PRE_INIT:
        state_info = FNC_TblAllocCtx(sizeof(local_ctx));
        sprintf (tbuf, "\n In Pre Init");
        trace(tbuf);
        break;
    case TBL_INIT:
        state_info = FNC_TblGetCtx();
        state_info->cnt=1;
        sprintf (tbuf, "\n In Init");
        trace(tbuf);
        break;
    case TBL_BUILD:
        state_info = FNC_TblGetCtx();
        if (state_info->cnt == 0 )
        {
            sprintf (tbuf, "\n In Build setting EOF, input 1 is %d", *in1);
            trace(tbuf);
            strcpy(sqlstate, "02000");
            return;
        }
        sprintf (tbuf, "\n In Build, input 1 is %d", *in1);
        trace(tbuf);
        *out1 = *in1;
        strcpy((char*)out2, (char*)in2);
        state_info->cnt = 0;
        break;
    case TBL_FINI:
        sprintf (tbuf, "\n In Fini");
        trace(tbuf);
        break;
    case TBL_END:
        sprintf (tbuf, "\n In END");
        trace(tbuf);
        break;
}

```

```

    }
}

```

## Example Output

```

SELECT * FROM table (phase (t.v, 'mike')) AS d
; SELECT * FROM udftrace ORDER BY 1,2;

```

\*\*\* Query completed. 10 rows found. 2 columns returned.

\*\*\* Total elapsed time was 1 second.

o1 o2

```

-----
5 mike
9 mike
7 mike
6 mike
3 mike
10 mike
1 mike
4 mike
8 mike
2 mike

```

\*\*\* Query completed. 52 rows found. 3 columns returned.

Amp\_number      Sequence Message

```

-----
0000           1   In Pre Init
0000           2   In Init
0000           3   In Build, input 1 is 5
0000           4   In Build setting EOF, input 1 is 5
0000           5   In Fini
0000           6   In Init
0000           7   In Build, input 1 is 1
0000           8   In Build setting EOF, input 1 is 1
0000           9   In Fini
0000          10   In Init
0000          11   In Build, input 1 is 8
0000          12   In Build setting EOF, input 1 is 8
0000          13   In Fini
0000          14   In Init
0000          15   In Build, input 1 is 2

```

```

0000      16   In Build setting EOF, input 1 is 2
0000      17   In Fini
0000      18   In END
0100        1   In Pre Init
0100        2   In Init
0100        3   In Build, input 1 is 9
0100        4   In Build setting EOF, input 1 is 9
0100        5   In Fini
0100        6   In END
0200        1   In Pre Init
0200        2   In Init
0200        3   In Build, input 1 is 7
0200        4   In Build setting EOF, input 1 is 7
0200        5   In Fini
0200        6   In Init
0200        7   In Build, input 1 is 4
0200        8   In Build setting EOF, input 1 is 4
0200        9   In Fini
0200       10   In END
0500        1   In Pre Init
0500        2   In Init
0500        3   In Build, input 1 is 6
0500        4   In Build setting EOF, input 1 is 6
0500        5   In Fini
0500        6   In END
0600        1   In Pre Init
0600        2   In Init
0600        3   In Build, input 1 is 3
0600        4   In Build setting EOF, input 1 is 3
0600        5   In Fini
0600        6   In END
0700        1   In Pre Init
0700        2   In Init
0700        3   In Build, input 1 is 10
0700        4   In Build setting EOF, input 1 is 10
0700        5   In Fini
0700        6   In END

```

## C Constant/Variable Mode Table Function

This example shows the C code for a table function that supports both TBL\_MODE\_CONST and the TBL\_MODE\_VARY modes. The TBL\_MODE\_CONST is very simple in that any one AMP will extract the data out of the text string that is passed in.

This example extracts some data out of a 'raw' text field to generate a bunch of rows. The raw data is in this format:

```
store_number,entries:entry[;...]
```

#### **entry**

```
customer_ID,item_ID
```

#### **store\_number**

Number that identifies the store that sold the items to the customers.

#### **entries**

Number of items sold.

#### **customer\_ID**

Customer identifier.

#### **item\_ID**

Item identifier.

## SQL Definition

```
CREATE FUNCTION extract_field(Text VARCHAR(32000),
                             From_Store INTEGER)
RETURNS TABLE (Customer_ID INTEGER,
                Store_ID     INTEGER,
                Item_ID       INTEGER)

LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME extract_field;
```

## Example Query

Here is a sample of how to invoke the table function with constant expression input arguments:

```
SELECT *
FROM TABLE (extract_field('25,2:9005,7896,9004,7839;36,1:737,9387;',
25)) AS t1;
```

Here is a sample of how to invoke the table function using the columns from a derived table as input arguments:

```
SELECT DISTINCT cust.Customer_ID, cust.Item_ID
FROM raw_cust,
     TABLE (extract_field(raw_cust.pending_data, 25))
AS cust
WHERE raw_cust.region = 1;
```

Here is the definition of the raw\_cust table:

```
CREATE SET TABLE RGS.raw_cust ,NO FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT
     (
       region INTEGER,
       pending_data VARCHAR(32000) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX ( region );
```

Here is a sample of the data in the raw\_cust table:

```
region pending_data
```

```
-----
2 7,2:879,3788,879,4500,390,9004;08,1:500,9056;
1 25,3:9005,3789,9004,4907,398,9004;36,2:738,9387,738,9550;
1 25,2:9005,7896,9004,7839;36,1:737,9387;
```

## C Function Definition

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

/*****/
/* The definition of the scratch pad */
/*****/
typedef struct
{
    int  custid;
    int  itemid;
} item_t;
```

```

typedef struct {
    int      Num_Items;
    int      Cur_Item;
    INTEGER   store_num;
    item_t    *Item_List;
} local_ctx;

/*****
/* This copy to SQL_TEXT fields will work to copy ASCII strings to */
/* SQL_TEXT strings (in this case error_message) for any character */
/* set mode. In other words if the SQL_TEXT is defined as */
/* Unicode_Text it will still work to give the proper error message */
*****/
static void unicpy(SQL_TEXT *dest,
                   char *src)

{
    while (*src)
        *dest++ = *src++;
}

/*****
/* A simple function to scan to the next break in the text based on */
/* the delimiter passed in */
*****/
static VARCHAR_LATIN *next(VARCHAR_LATIN find,
                           VARCHAR_LATIN *data)
{
    while (*data != '\0')
    {
        if (*data == find)
            break;
        data++;
    }
    return data;
}

/*****
/* The text data that this function processes is in a very simple */
/* format: */
/* */
/* <storenum>,<num items>:<customer id>,<item number>, ... ; */
/* <storenum>,<num items>: ... */
*****/

```



```

/*****
/* Do a pre-scan of the text and save the data. Note: This pre-scan */
/* routine actually extracts all needed data out of the text field */
/* and saves it in allocated memory via FNC_malloc. With this logic */
/* when it gets to the TBL_BUILD phase the data will simply be taken */
/* from the saved area. There is no need to look at the original */
/* string again during the TBL_BUILD phase. This is just one way to */
/* design it. The alternative is to just do the scanning each time */
/* TBL_BUILD is called from the text data field that is passed in */
/* to the table function at all times. It is a choice that the */
/* designer must make when developing the application. */
*****/
static int Prescan(local_ctx      *info,
                   VARCHAR_LATIN *Text,
                   INTEGER         *frmstore)
{
    INTEGER storenum;
    int i;
    int num_items = 0;
    VARCHAR_LATIN *Tscan = Text;

    /* find the data for the store we are interested in */
    while (*Tscan )
    {
        sscanf((char *) Tscan, "%d", &storenum);
        if (*frmstore == storenum)
        {
            /* found the entry of interest - get the information */
            /* on how many items there are */
            Tscan = next(',', Tscan)+1;
            sscanf((char *) Tscan, "%d", &num_items);
            break;
        }
        /* find next store */
        Tscan = next(';', Tscan);
        if (*Tscan == '\0')
            break;
        Tscan++;
    }

    /* let's malloc some worst case memory to keep track of the items */
    /* we collect */

```

```

if (num_items)
{
    info->Item_List = FNC_malloc(sizeof(item_t)*num_items);
    if (info->Item_List == NULL)
        /* not good - should have been able to get the memory */
        return -1;
}
else
{
    info->Num_Items = 0;
    return 0;
}

/* now let's find all the entries for the store that we are */
/* interested in */
/* skip to first item */
Tscan = next(':', Tscan)+1;
for (i=0; i<num_items; i++)
{
    sscanf((char *) Tscan,
           "%d,%d",
           &info->Item_List[i].custid,
           &info->Item_List[i].itemid);
    Tscan = next(',', Tscan)+1;
    Tscan = next(',', Tscan)+1;
}

info->Num_Items = num_items;
info->store_num = *frmstore;
return num_items;
}

/*****
/* Extract all of the data now. Actually this routine just takes */
/* the items that Prescan built and transfers the data out one item */
/* at a time. Notice that it does not build the output column if it */
/* is not being asked for. For this simple example it probably */
/* makes no difference, but if there is a lot of complexity in the */
/* application to build some columns then it could when noticing */
/* that a field is null not go through the computation to build it */
/* at all. In fact the Prescan function could have been smarter and */
/* not built the list of fields that are not being asked for */
*****/
static int Extract(local_ctx      *info,
                  INTEGER          *custid,

```

```

        INTEGER      *store,
        INTEGER      *itemid,
        int          custid_i,
        int          store_i,
        int          item_i)
{
    /* check to see if there is something left to extract */
    if (info->Cur_Item == info->Num_Items)
        return 0;

    /* okay let's set the output data only if they want it */
    if (custid_i == 0)
        *custid = info->Item_List[info->Cur_Item].custid;
    if (store_i == 0)
        *store = info->store_num;
    if (item_i == 0)
        *itemid = info->Item_List[info->Cur_Item].itemid;

    /* set up for next item the next time */
    info->Cur_Item++;
    return 1;
}

/*****
/* Do a reset of the context block */
*****/
static void Reset(local_ctx *info)
{
    info->Num_Items = 0;
    info->Cur_Item = 0;
    info->Item_List = NULL;
}

/*****
/* Clean up upon error or when done. Needs to free up any memory
/* that was allocated or it will return an error message. Note that
/* the memory was allocated outside of the general scratch pad. But
/* the address must be retained in the scratch pad or you have no
/* means of referencing the data or freeing it for subsequent calls. */
*****/
static void Clean_Up(local_ctx *info)
{
    if (info->Item_List)
        FNC_free(info->Item_List);
}

```

```

}
void extract_field(VARCHAR_LATIN *Text, /* field decode */
                  INTEGER *frmStore, /* data to extract */
                  INTEGER *custid, /* 1st output column for row */
                  INTEGER *store, /* 2nd output column */
                  INTEGER *item,
                  int *Text_i, /* in parameter indicator */
                  int *frmstore_i, /* if no store, return no row */
                  int *custid_i, /* 1st output indicator for */
                  int *store_i, /* row, and so on */
                  int *item_i,
                  char sqlstate[6],
                  SQL_TEXT fncname[129],
                  SQL_TEXT sfncname[129],
                  SQL_TEXT error_message[257] )
{
    local_ctx *state_info;
    FNC_Phase Phase;
    int status;

    /* make sure the function is called in the supported context */
    switch (FNC_GetPhase(&Phase))
    {
        /******
        /* Process the constant expression case. Only one AMP will */
        /* participate for this example */
        /******
        case TBL_MODE_CONST:

            /* depending on the phase decide what to do */
            switch(Phase)
            {
                case TBL_PRE_INIT:
                    /* let the system know that I want to be the participant */
                    switch (FNC_TblFirstParticipant() )
                    {
                        case 1: /* participant */
                            return;

                        case 0: /* not participant */
                            /* don't participate */
                            if (FNC_TblOptOut())
                            {

```

```

        strcpy(sqlstate, "U0006"); /* an error return */
        unicpy(error_message, "Opt-out failed.");
        return;
    }
    break;
default: /* -1 or other error */
    strcpy(sqlstate, "U0007");
    unicpy(error_message,
           "First Participant logic did not work");
    return;
}

case TBL_INIT:
    /* get scratch memory to keep track of things */
    state_info = FNC_TblAllocCtx(sizeof(local_ctx));
    Reset(state_info);
    /* Preprocess the Text */
    status = Prescan(state_info, Text, frmStore );
    if (status == -1)
    {
        Clean_Up(state_info);
        strcpy(sqlstate, "U0008");
        unicpy(error_message, "Text had pre-scan errors.");
        return;
    }

    break;

case TBL_BUILD:
    state_info = FNC_TblGetCtx();

    status = Extract(state_info,
                     custid,
                     store,
                     item,
                     *custid_i,
                     *store_i,
                     *item_i);
    if (status == 0)
        /* Have no more data, return no data sqlstate. */
        strcpy(sqlstate, "02000");
    else if (status == -1)
    {
        Clean_Up(state_info);

```

```

        strcpy(sqlstate, "U0009");
        unicpy(error_message, "Text had extract error.");
        return;
    }
    break;
case TBL_END:
    /* everyone done */
    state_info = FNC_TblGetCtx();
    Clean_Up(state_info);
    break;
}
break;

/*****
/* Process the varying expression case. */
*****/
case TBL_MODE_VARY:
    switch(Phase)
    {
    case TBL_PRE_INIT:
        /* get scratch memory to use from now on */
        state_info = FNC_TblAllocCtx(sizeof(local_ctx));
        Reset(state_info);
        break;

    case TBL_INIT:
        /* Preprocess the Text */
        state_info = FNC_TblGetCtx();
        status = Prescan(state_info, Text, frmStore );
        if (status == -1)
        {
            status = FNC_TblAbort();
            if (status == 1)
            {
                Clean_Up(state_info);
                strcpy(sqlstate, "U0008");
                unicpy(error_message, "Text had pre-scan errors");
                return;
            }
        }
        break;

    case TBL_BUILD:
        state_info = FNC_TblGetCtx();

```

```

        status = Extract(state_info,
                        custid,
                        store,
                        item,
                        *custid_i,
                        *store_i,
                        *item_i);
    if (status == 0)
        /* Have no more data; return no data sqlstate. */
        strcpy(sqlstate, "02000");
    else if (status == -1)
    {
        status = FNC_TblAbort();
        /* If I was the first then report the error */
        if (status = 1)
        {
            Clean_Up(state_info);
            strcpy(sqlstate, "U0009");
            unicpy(error_message, "Text had extract error");
        }
        return;
    }
    break;
case TBL_FINI:
    /* Initialize for the next set of data. */
    state_info = FNC_TblGetCtx();
    Clean_Up(state_info);
    Reset(state_info);
    break;
case TBL_END:
    /* Everyone done. */
    state_info = FNC_TblGetCtx();
    Clean_Up(state_info);
    break;
case TBL_ABORT:
    state_info = FNC_TblGetCtx();
    Clean_Up(state_info);
    break;
    }
}
}

```

## C Constant/Variable Mode Table Function With Dynamic Result Row Specification

This example is similar to the preceding example, [C Constant/Variable Mode Table Function](#), and shows the C code for a table function that supports both TBL\_MODE\_CONST and the TBL\_MODE\_VARY modes. The TBL\_MODE\_CONST is very simple in that any one AMP will extract the data out of the text string that is passed in.

The table function can return a maximum of four columns, as specified by the RETURNS TABLE VARYING COLUMNS clause in the REPLACE FUNCTION statement.

Each output parameter in the table function parameter list is declared as a void pointer because the actual data types of the result row arguments are unknown until function invocation.

During execution, the table function calls the FNC\_TblGetColDef library function to get the actual number and data types of the result row arguments.

### SQL Definition

```
REPLACE FUNCTION get_data (Text VARCHAR(100))
RETURNS TABLE VARYING COLUMNS(4)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!dynamic!udfsrc/dynamic.c!f!get_data';
```

### Example Query

Here is an example of how to invoke the table function with constant expression input arguments and get a result row with four columns:

```
SELECT *
FROM TABLE (get_data('1356,122,20061009,abc'))
RETURNS (col1 INTEGER, col2 BYTEINT, col3 DATE, col4 VARCHAR(5))
AS t1;
```

Here is another example of how to invoke the table function with constant expression input arguments and get a result row with three columns:

```
SELECT *
FROM TABLE (get_data('1234,56,20061010'))
RETURNS (col1 INTEGER, col2 BYTEINT, col3 DATE)
AS t1;
```



Here is an example of how to invoke the table function using the columns from a derived table as input arguments:

```
SELECT tf.col1, tf.col2, tf.col3, tf.col4
FROM raw_cust,
     TABLE (get_data(raw_cust.text)
RETURNS (col1 INTEGER, col2 BYTEINT, col3 DATE, col4 VARCHAR(5)))
AS tf;
```

Here is the definition of the raw\_cust table:

```
CREATE TABLE raw_cust (C1 INT, text VARCHAR(30));

INSERT INTO raw_cust (1,'1356,122,20061009,abc');
```

## C Function Definition

```
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <stdio.h>
#include <string.h>

/* definition of the scratch pad */

#define FALSE    0
#define TRUE     1
#define MAX_OUT_PARAMS 4
typedef struct
{
    int indicator;
    char dtype[5];
    void *value;
} column_info;

typedef struct
{
    int done;
    VARCHAR_LATIN *Tscan;
    column_info cols[MAX_OUT_PARAMS];
} gen_ctx;

/******/
```

```

/* A simple function to scan the next break in the text based on */
/* the delimiter passed in.                                     */
/*****
static VARCHAR_LATIN *
next(VARCHAR_LATIN find,VARCHAR_LATIN *data)
{
    while(*data!='\0') {
        if(*data==find)
            break;
        data++;
    }
    return data;
}

/*****
/* Obtain all result column definitions for a table function with */
/* variable output columns.                                     */
/*****
static int
CheckColDef(gen_ctx *info,FNC_ColumnDef_t *col_def,
            VARCHAR_LATIN *Text,
            void *col1,void *col2,void *col3,void *col4,
            int col1_i,int col2_i,int col3_i, int col4_i)
{
    int j;
    int i;
    VARCHAR_LATIN *TScan=Text ;
    info->Tscan=TScan;

    // Prepare output datatype string format for sscanf
    for(j=0;j<col_def->num_columns;j++)
    {

        switch(col_def->column_types[j].datatype)
        {
            case DATE_DT:
            case INTEGER_DT:
            case BYTEINT_DT:
            case SMALLINT_DT:
                strcpy(info->cols[j].dtype,"%d");
                break;

            case CHAR_DT:
                strcpy(info->cols[j].dtype,"%c");

```

```

        break;

    case VARCHAR_DT:
        strcpy(info->cols[j].dtype,"%s");
        break;
    default:
        return -1;
    }
}

info->cols[0].indicator=col1_i;
info->cols[1].indicator=col2_i;
info->cols[2].indicator=col3_i;
info->cols[3].indicator=col4_i;
info->cols[0].value=col1;
info->cols[1].value=col2;
info->cols[2].value=col3;
info->cols[3].value=col4;

/* find all output columns that specified in sel stmt */
for(i=0;i<col_def->num_columns;i++)
{
    if(info->cols[i].indicator==0)
    {
        sscanf((char *)info->Tscan,info->cols[i].dtype,
            info->cols[i].value);
        info->Tscan=next(',',info->Tscan)+1;
    }
}
return 0;
}

/*****
/* This is a simple demo program to show how to utilize the result */
/* returned from FNC_TBLGetColDef(). */
/*****/
void
get_data (VARCHAR_LATIN *Text,    /* input string */
          void          *col1,    /* output parameters */
          void          *col2,
          void          *col3,
          void          *col4,
          int           *Text_i,  /* input parameter indicator */
          int           *col1_i,  /* output indicators for row ...*/
          int           *col2_i,

```

```

        int          *col3_i,
        int          *col4_i,
        char          sqlstate[6],
        SQL_TEXT      fncname[129],
        SQL_TEXT      sfncname[129],
        SQL_TEXT      error_message[257] )
{
    int status;
    FNC_ColumnDef_t *The_Columns;
    FNC_Phase Phase;
    gen_ctx *LocalGenCtx;

    /* Definition of result parameters returned from FNC_TblGetColDef */
    The_Columns = FNC_TblGetColDef();

    /* this will keep the state of which rows have been sent */
    /* make sure the function is called in the supported context */
    switch (FNC_GetPhase(&Phase))
    {
        /******
        /* Process the constant expression case. Only one AMP */
        /* participates for this example.                      */
        /******
        case TBL_MODE_CONST:

            /* Depending on the phase, decide what to do. */
            switch(Phase)
            {
                case TBL_PRE_INIT:
                    switch (FNC_TblFirstParticipant() )
                    {
                        case 1:      /* participant */
                            return;
                        case 0:      /* not participant */
                            /* Don't participate. */
                            if (FNC_TblOptOut())
                            {
                                strcpy(sqlstate, "U0006"); /* Return error. */
                                strcpy((char *) error_message,"Opt-out failed.");
                                return;
                            }
                        break;
                        default: /* -1 or other error */
                            strcpy(sqlstate, "U0007");
                    }
            }
    }

```

```

        strcpy((char *) error_message,
            "First Participant Logic did not work.");
        return;
    }

case TBL_INIT:
    /* Allocate a general scratchpad to retain data */
    /* between iterations. */
    LocalGenCtx = FNC_TblAllocCtx(sizeof(gen_ctx));
    LocalGenCtx->done = FALSE;
    break;

case TBL_BUILD:
    /* Get the scratchpad. */
    LocalGenCtx = FNC_TblGetCtx();

    if (LocalGenCtx->done)
    {
        /* Have no more data; return no data sqlstate. */
        strcpy(sqlstate, "02000");
        strcpy((char *) error_message, "EOF");
        return;
    }

    /* Check and prepare output column datatype */
    status = CheckColDef (LocalGenCtx, The_Columns, Text,
                        col1, col2, col3, col4,
                        *col1_i, *col2_i, *col3_i, *col4_i);
    if (status == -1)
    {
        status = FNC_TblAbort();
        /* If I was the first then let's report the error. */
        strcpy(sqlstate, "U0009");
        return;
    }

    LocalGenCtx->done = TRUE;
    break;
case TBL_END:
case TBL_ABORT:
    break;
}

break;

```

```

/*****
/* Process the varying expression case. */
*****/
case TBL_MODE_VARY:
    switch(Phase)
    {
        case TBL_PRE_INIT:
            LocalGenCtx = FNC_TblAllocCtx(sizeof(gen_ctx));
            if (LocalGenCtx == NULL)
            {
                strcpy (sqlstate, "U006");
                strcpy ((char *) error_message,
                    "Context Allocation failed.");
                return;
            }
            break;
        case TBL_INIT:
            LocalGenCtx = FNC_TblGetCtx();
            LocalGenCtx->done = FALSE;
            break;
        case TBL_BUILD:
            LocalGenCtx = FNC_TblGetCtx();
            if (LocalGenCtx->done)
            {
                /* Have no more data return no data sqlstate */
                strcpy(sqlstate, "02000");
                strcpy((char *) error_message, "EOF");
                return;
            }

            status = CheckColDef (LocalGenCtx, The_Columns, Text,
                                col1, col2, col3, col4,
                                *col1_i, *col2_i, *col3_i, *col4_i);

            if (status == -1)
            {
                status = FNC_TblAbort();
                /* if I was the first then let's report the error */
                strcpy(sqlstate, "U0009");
                return;
            }

            LocalGenCtx->done = TRUE;
            break;
        case TBL_END:

```

```

        case TBL_ABORT:
            break;
    }
}
}

```

## C Table Operator

This example shows the C code for a table operator.

The required inputs for this aggregation operator are:

- The input table (or table expression) specified in the FROM clause of the SELECT statement. For this example, we will use the following table and input:

```

CREATE TABLE tab1 (A INTEGER,
                    B INTEGER,
                    C INTEGER,
                    D INTEGER);

INSERT INTO tab1(1,2,3,4);
INSERT INTO tab1(2,3,4,5);

```

- Each aggregate is a Custom clause, such as SUM\_AGG(c1, c2) and AVG\_AGG(c3).

## SQL Definition

```

REPLACE FUNCTION udaggregation ()
RETURNS TABLE VARYING USING FUNCTION udaggregation_contract
LANGUAGE C
NO SQL
PARAMETER STYLE SQLTABLE
EXTERNAL NAME 'CS!udaggregation!udaggregation.c!F!udaggregation!D';

```

## Example Query

```

SELECT *
FROM udaggregation (
    on (select 1 as pa, 2 as pa2, tab1.* from tab1)
    local order by pa ASC NULLS FIRST, pa2 DESC NULLS LAST
    using SUM_AGG('A', 'B')
    AVG_AGG('C')
) AS D;

```

```

SELECT *
FROM udaggregation (
on (select 1 as pa, tab1.* from tab1)
partition by pa
using SUM_AGG('A', 'B')
AVG_AGG('C')
) AS D;

```

## C Function Definition for the Contract Function

This example shows the C code for the interface function for the parser.

```

#define byte unsigned char
#define boolean int
#define false 0
#define true 1
#define OK 0
#define SQL_TEXT Latin_Text

#define _CRT_SECURE_NO_DEPRECATE
#include <limits.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqltypes_td.h>

#define FALSE 0
#define TRUE 1

#define MAX_KEY_LEN 100

void udaggregation_contract (
    INTEGER *result,
    int *indicator_Result,
    char sqlstate[6],
    SQL_TEXT extname[129],
    SQL_TEXT specific_name[129],
    SQL_TEXT error_message[257])
{
    FNC_TblOpColumnDef_t *Output_columns; // column definition for output stream
    FNC_TblOpColumnDef_t *Input_columns; // column definition for input stream
    char
        *buf;

```



```

int                colcount;           // number of columns in output
int                i, j, n, k;
char               key[MAX_KEY_LEN]; // key in custom clause
int                keylen;             // actual key length
Key_info_t        valuesSUM;          // values associated with key SUM
Key_info_t        valuesAVG;          // values associated with key AVG
int                *index;             // indices of columns in input stream to
                                      // apply aggregates

boolean_t          found;
char               colname[FNC_MAXNAMELEN_EON];

/* ----- process custom clause information ----- */
FNC_TblOpGetCustomKeyInfoOf("SUM_AGG", &valuesSUM);
FNC_TblOpGetCustomKeyInfoOf("AVG_AGG", &valuesAVG);

// allocate space for values associated with keys "SUM_AGG" and "AVG_AGG"
valuesSUM.values_r = FNC_malloc( sizeof(Values_t) * valuesSUM.numOfVal );
valuesAVG.values_r = FNC_malloc( sizeof(Values_t) * valuesAVG.numOfVal );

// compute number of columns in output stream
colcount = valuesSUM.numOfVal + valuesAVG.numOfVal;

// get values associated with key "SUM_AGG"
FNC_TblOpGetCustomValuesOf(&valuesSUM);

// get values associated with key "AVG_AGG"
FNC_TblOpGetCustomValuesOf(&valuesAVG);

/* ----- define column definition for output stream -----*/

// Allocate memory for the output columns
Output_columns = FNC_malloc ( TblOpSIZECOLDEF(colcount) );

// initialize output columns
TblOpINITCOLDEF(Output_columns, colcount);

// Allocate memory for input columns
Input_columns = FNC_malloc ( TblOpSIZECOLDEF( FNC_TblOpGetColCount(0, 'R') ) );

// initialize input columns
TblOpINITCOLDEF( Input_columns, FNC_TblOpGetColCount(0, 'R') );
FNC_TblOpGetColDef(0, 'R', Input_columns);

```

```

// allocate space for indices of columns in input stream to apply aggregates
index = FNC_malloc( sizeof(int) * colcount );

/* ----- Fill in column information for SUM_AGG ----- */

// Fill in column information for output stream for SUM_AGG
k = 0;
found = false;
for (i = 0; i < valuesSUM.numOfVal; i++)
{
    // check value corresponds to a column name in input stream
    for (j = 0; j < Input_columns->num_columns; j++)
    {
        if (strncmp(valuesSUM.values_r[i].value,
                    Input_columns->column_types[j].column,
                    valuesSUM.values_r[i].valueLen) == 0)
        {
            index[k] = j;
            k++;
            found = true;
            break;
        }
    }
}
if (! found)
{
    FNC_TblOpSetError("U0001", "Invalid column name in custom clause.");
    return;
}
strcpy(colname, "SUM_");
strncpy( Output_columns->column_types[i].column,
        strncat(colname, (char *) valuesSUM.values_r[i].value,
                valuesSUM.values_r[i].valueLen), FNC_MAXNAMELEN_EON);
Output_columns->column_types[i].datatype = REAL_DT;
Output_columns->column_types[i].bytesize = sizeof_FLOAT;
}

n = valuesSUM.numOfVal;
// Fill in column information for output stream for AVG_AGG
found = false;
for (i = 0; i < valuesAVG.numOfVal; i++)
{
    // check value corresponds to a column name in input stream
    for (j = 0; j < Input_columns->num_columns; j++)
    {

```

```

        if (strncmp(valuesAVG.values_r[i].value,
                    Input_columns->column_types[j].column,
                    valuesAVG.values_r[i].valueLen) == 0)
        {
            index[k] = j;
            k++;
            found = true;
            break;
        }
    }
    if (! found)
    {
        FNC_TblOpSetError("U0001", "Invalid column name in custom clause.");
        return;
    }
    strcpy(colname, "AVG_");
    strncpy( Output_columns->column_types[n+i].column,
            strncat(colname, (char *) valuesAVG.values_r[i].value,
                    valuesAVG.values_r[i].valueLen), FNC_MAXNAMELEN_EON);
    Output_columns->column_types[n+i].datatype = REAL_DT;
    Output_columns->column_types[n+i].bytesize = sizeof_FLOAT;
}
// set column definitions for output stream
FNC_TblOpSetOutputColDef(0, Output_columns);

// pass indices to table operator in the contract context
FNC_TblOpSetContractDef(index, sizeof(int) * colcount );

// release memory
*result = Output_columns->num_columns;
FNC_free(Output_columns);
FNC_free(Input_columns);
FNC_free(valuesSUM.values_r);
FNC_free(valuesAVG.values_r);
FNC_free(index);
}

```

## C Function Definition for the Table Operator

```

void udaggregation ()
{
    int                null_ind, length;
    FNC_TblOpHandle_t  *Input_handle;    // input stream handle

```

```

FNC_TblOpHandle_t      *Output_handle;      // output stream handle
FNC_TblOpColumnDef_t   *Input_columns;      // input stream column definitions
FNC_TblOpColumnDef_t   *Output_columns;     // output stream column definitions
double                *aggregates;          // aggregates computation
int                   *index;               // columns index in input stream
of aggregates
int                   rowcount = 0 ;        // number of rows
int                   i, j, k, tmp;
BYTE                  *ptr;
int                   colcount;

/* Allocate memory for the output columns */
colcount = FNC_TblOpGetColCount(0, 'W');
Output_columns = FNC_malloc ( TblOpSIZECOLDEF( colcount ) );
/* initialize output columns */
TblOpINITCOLDEF(Output_columns, colcount);
FNC_TblOpGetColDef(0, 'W', Output_columns);

/* Allocate memory for input columns */
Input_columns = FNC_malloc ( TblOpSIZECOLDEF( FNC_TblOpGetColCount(0, 'R') ) );
/* initialize input columns */
TblOpINITCOLDEF( Input_columns, FNC_TblOpGetColCount(0, 'R') );
FNC_TblOpGetColDef(0, 'R', Input_columns);

/* initialize aggregated values for group */
aggregates = FNC_malloc( sizeof(float) * Output_columns->num_columns );
for (i=0; i<Output_columns->num_columns; i++)
{
    aggregates[i] = 0;
}
/* get indices from contract context */
index = FNC_malloc( FNC_TblOpGetContractLength() );
FNC_TblOpGetContractDef(index, FNC_TblOpGetContractLength(), &tmp );
/* The basic row iterator would be structured as follows */

Input_handle = FNC_TblOpOpen(0, 'R', TBLOP_NOOPTIONS); // start iterator for
input stream
Output_handle = FNC_TblOpOpen(0, 'W', TBLOP_NOOPTIONS); // start iterator for
output stream

while ( FNC_TblOpRead(Input_handle) == TBLOP_SUCCESS)
{
    rowcount++;
    // update aggregate for each column

```

```

    for (i=0; i<Output_columns->num_columns; i++)
    {
        /* increment aggregated values */
        FNC_TblOpGetAttributeByNdx(Input_handle, index[i], (void **) &ptr,
&null_ind, &length);
        switch (Input_columns->column_types[index[i]].datatype)
        {
            case BYTEINT_DT: aggregates[i] += *((signed char *) ptr); break;
            case SMALLINT_DT: aggregates[i] += *((short *) ptr); break;
            case INTEGER_DT: aggregates[i] += *((int *) ptr); break;
            case BIGINT_DT: aggregates[i] += *((long long *) ptr); break;
        }
    }
}

/* set output values */
for (i=0; i<Output_columns->num_columns; i++)
{
    if (Output_columns->column_types[i].column[0] == 'A')
    {
        if (rowcount != 0)
        {
            aggregates[i] = aggregates[i] / ((double) rowcount);
        }
    }
    FNC_TblOpBindAttributeByNdx(Output_handle, i, aggregates+i, 0, sizeof(double));
}

/* write output row */
FNC_TblOpWrite(Output_handle);

FNC_TblOpClose(Input_handle);
FNC_TblOpClose(Output_handle);

// release memory
FNC_free(Output_columns);
FNC_free(Input_columns);
FNC_free(aggregates);
FNC_free(index);
}

```

## C Function Using GLOP Data

This example shows how to create a GLOP set, add GLOP data to the GLOP set, create a UDF as a member of the GLOP set, and use the GLOP data from the UDF. Take the following steps to use this example:

1. If the DBCExtension database does not exist, use the DIPGLOP script to create it.  
For information on the DIP utility and the DIPGLOP script, see *Teradata Vantage™ - Database Utilities*, B035-1102.
2. Use the CREATE GLOP SET statement to create a GLOP set called MyGlopTestSet.  
See [SQL Statement to Create GLOP Set](#).
3. Create a UDF called compinform that returns a BLOB that will be used as the GLOP data.  
See [SQL Definition for the Function That Generates GLOP Data](#) and [C Function Definition That Generates GLOP Data](#).
4. Call DBCExtension.GLOP\_Add to add the GLOP data to the GLOP set.  
See [Add Data to GLOP Set](#).
5. Create a UDF called glop\_map1 that is a member of the GLOP set and maps the GLOP set to use the data.  
See [SQL Definition for the Function That Uses GLOP Data](#) and [C Function Definition That Uses GLOP Data](#).
6. Call the glop\_map1 UDF.  
See [Example Query That Calls the UDF That Uses the GLOP Data](#).

## SQL Statement to Create GLOP Set

```
CREATE GLOP SET MyGlopTestSet;
```

## Add Data to GLOP Set

```
-- Adding data to GLOP Set <MyGlopTestSet>

CALL DBCExtension.GLOP_Add
  ('MyGlopTestSet', 'SY', NULL, 'compdatainfo1',
   'N', 0, 'N', 'W', 'E', 0, -1, 1, compinform());
```

## SQL Definition for the Function That Generates GLOP Data

Here is the SQL statement for the function that generates GLOP data.

```

REPLACE FUNCTION compinform()
RETURNS BLOB AS LOCATOR
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!compinform!compinform.c';

```

## SQL Definition for the Function That Uses GLOP Data

Here is the SQL statement for the function that reads GLOP data. The USING GLOP SET clause makes the UDF a member of the MyGlopTestSet GLOP set.

```

REPLACE FUNCTION glop_map1(record_n INTEGER)
RETURNS VARCHAR(30)
LANGUAGE C
USING GLOP SET MyGlopTestSet
NO SQL
PARAMETER STYLE SQL
EXTERNAL NAME 'CS!glop_map1!glop_map1.c';

ALTER FUNCTION glop_map1 execute not protected;

```

## Example Query That Calls the UDF That Uses the GLOP Data

```

SELECT glop_map1(0);

```

## C Function Definition That Generates GLOP Data

This function generates GLOP data that is added to a GLOP set as a system type GLOP.

```

/***** File: compinform.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdlib.h>
#define IsNull -1
#define IsNotNull 0
#define loopsize 20

/*****
/* Generate data for different departments in a retail store. */

```

```

/*****/

struct Company
{
    INTEGER deptid;
    char deptname[30];
};

struct Company_INFO
{
    struct Company Dept[loopsize];
}*COMPANY_DETAILS;

void compinform( LOB_RESULT_LOCATOR *cmp_info,
                INTEGER *cmp_infoISNULL,
                char sqlstate[6],
                SQL_TEXT extname[129],
                SQL_TEXT specific_name[129],
                SQL_TEXT error_message[257] )
{
    FNC_LobLength_t actlen;
    int i;
    const char truncate1a[6] = "22001";
    char *Dept_Name[20] =
        {"Automotive", "Hardware", "Bed", "Bath", "Toys", "Food",
         "Electronics", "Games", "Photo", "Kitchen", "Seasonal", "Books",
         "Furniture", "Pharmacy", "Men", "Women", "Baby", "Kids",
         "Shoes", "Nursery"};

    // Allocate memory to store the contents.

    COMPANY_DETAILS =
        (struct Company_INFO *)FNC_malloc(sizeof(struct Company_INFO));

    // Store data in a structure format.

    for (i=0; i<loopsize; i++)
    {
        COMPANY_DETAILS->Dept[i].deptid=i+1;
        strcpy(COMPANY_DETAILS->Dept[i].deptname,(char*)Dept_Name[i]);
    }
}

```



```

// Add data to BLOB.

if (FNC_LobAppend(*cmp_info,COMPANY_DETAILS,
                 sizeof(*COMPANY_DETAILS),& actlen) !=0)
{
    strcpy(sqlstate, truncate1a);
    FNC_free(COMPANY_DETAILS);
    return;
}

// Free the memory that was allocated to the structure CompanyName.

FNC_free(COMPANY_DETAILS);
}

```

## C Function Definition That Uses GLOP Data

This function maps the GLOP set and then reads the GLOP data.

```

/***** File: glop_map1.c *****/
#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>
#include <stdlib.h>
#define IsNull -1
#define IsNotNull 0
#define LOOPSIZE 20

/*****
/* Function to read GLOP data for the given record_n number. */
*****/

struct Company
{
    INTEGER deptid;
    char deptname[30];
};

struct Company_Info
{
    struct Company Dept[LOOPSIZE];
}*COMPANY_DETAILS;

```

```

void glop_map1( INTEGER *record_n,
               VARCHAR_LATIN *Deptname1,
               INTEGER *record_nIsNull,
               INTEGER *Deptname1IsNull,
               char sqlstate[6],
               SQL_TEXT extname[129],
               SQL_TEXT specific_name[129],
               SQL_TEXT error_message[257] )
{
    int glop_stat, MAP_ADD;
    int i;
    GLOP_Map_t *MyGLOP;

    if (*record_nIsNull == IsNull)
    {
        strcpy(sqlstate, "22004") ;
        strcpy((char *) error_message, "Null value not allowed.") ;
        *Deptname1IsNull = IsNull;
        return;
    }

    strcpy((char *) error_message, " ");
    *Deptname1IsNull = IsNotNull;

    MAP_ADD = 0; // Use the first GLOP section of the GLOP set.
    i = *record_n;
    glop_stat = FNC_Get_GLOP_Map(&MyGLOP);

    // Check whether mapping is established or not

    if (glop_stat)
    {
        if (glop_stat == -1)
            strcpy((char*)error_message,
                  "Map not set up. Exceeded GLOP data mapping limits.");
        else if (glop_stat == -2)
            strcpy((char*)error_message,
                  "Not member of GLOP set or no rows in GLOP tables.");
        strcpy(sqlstate,"U0001");
        return;
    }

    // Check whether specified row for the GLOP exists or not

```

```

if (MyGLOP->GLOP[MAP_ADD].GLOP_ptr == NULL)
{
    strcpy((char*)error_message,"Map does not exist at index 0.");
    strcpy(sqlstate,"U0002");
    return;
}

if (MyGLOP->GLOP[MAP_ADD].size == 0)
{
    strcpy((char*)error_message,"Nothing is mapped.");
    strcpy(sqlstate,"22023");
    return;
}

// Assign the mapped address of GLOP data to the structure.

COMPANY_DETAILS = MyGLOP->GLOP[MAP_ADD].GLOP_ptr;

// Read the data.
strcpy(Deptname1,COMPANY_DETAILS->Dept[i].deptname);
}

```

## Java Scalar Function

This example shows the Java code for a scalar UDF that calculates factorials.

## JAR File Registration

The following statements register the JAR file for the scalar UDF with the JUDF database, creating an identifier for the JAR file called JarUDF:

```

DATABASE JUDF;
CALL SQLJ.INSTALL_JAR('C:\udfsrc\UDFExample.jar','JarUDF',0);

```

## SQL Definition

```

DATABASE JUDF;
CREATE FUNCTION factorial
(x INTEGER)
RETURNS INTEGER
LANGUAGE JAVA

```

```

NO SQL
PARAMETER STYLE JAVA
CALLED ON NULL INPUT
EXTERNAL NAME 'JarUDF:UDFExample.fact(java.lang.Integer) returns
java.lang.Integer';

```

## Example Query

```

SELECT pname, factorial(pkey)
FROM pRecords
ORDER BY pname;

```

## Java Method Implementation

```

public class UDFExample {
    public static Integer fact( Integer x ) {
        if (x == null)
            return null;
        int x_t = x.intValue();
        if (x_t < 0)
            return new Integer(0);
        int factResult = 1;
        while (x_t > 1) {
            factResult = factResult * x_t;
            x_t = x_t - 1;
        }
        return new Integer(factResult);
    }
}

```

## Java Function Using TD\_ANYTYPE Parameters

This example shows the Java code for a UDF that uses a TD\_ANYTYPE parameter.

## SQL Definition

```

CREATE FUNCTION jtdany001(P1 TD_ANYTYPE)
RETURNS VARCHAR(100)
NO SQL
PARAMETER STYLE JAVA
LANGUAGE JAVA

```

```
EXTERNAL NAME 'ANYTYPE_JAR:AnytypeClass.jtdany001(java.lang.Object)
returns java.lang.String';
```

## Java Implementation

```
public class AnytypeClass {

    public static java.lang.String jtdany001(java.lang.Object p1) throws Exception
    {
        String returnStr;

        if(p1 instanceof java.lang.Integer)
        {
            returnStr = "Parameter type is Integer";
            return returnStr;
        }
        else
            if(p1 instanceof java.lang.Byte)
            {
                returnStr = "Parameter type is ByteInt";
                return returnStr;
            }
            else
                if(p1 instanceof java.lang.Short)
                {
                    returnStr = "Parameter type is Short";
                    return returnStr;
                }
                else
                    if(p1 instanceof java.lang.Long)
                    {
                        returnStr = "Parameter type is BigInt";
                        return returnStr;
                    }
                    else if(p1 instanceof java.lang.Double)
                    {
                        returnStr = "Parameter type is Float/Real";
                        return returnStr;
                    }
                    else if(p1 instanceof java.math.BigDecimal)
                    {

```

```

        returnStr = "Parameter type is Decimal/Numeric";
        return returnStr;
    }
    else if(p1 instanceof java.sql.Date)
    {
        returnStr = "Parameter type is Date";
        return returnStr;
    }
    else if(p1 instanceof java.sql.Time)
    {
        returnStr = "Parameter type is Time";
        return returnStr;
    }
    else if(p1 instanceof java.sql.Timestamp)
    {
        returnStr = "Parameter type is Timestamp";
        return returnStr;
    }
    else if(p1 instanceof java.sql.Clob)
    {
        returnStr = "Parameter type is Clob";
        return returnStr;
    }
    else if(p1 instanceof java.sql.Blob)
    {
        returnStr = "Parameter type is Blob";
        return returnStr;
    }
    else if(p1 instanceof java.lang.String)
    {
        returnStr = "Parameter type is Char/Varchar/Interval";
        return returnStr;
    }
    else
    {
        returnStr = "Invalid object type passed.";
        return returnStr;
    }
}

```

## Java Aggregate Function: Arithmetic Sum

This example shows the Java code for a simple aggregate UDF that calculates the arithmetic sum for a group.

### JAR File Registration

The following statements register the JAR file for the aggregate UDF with the JUDF database, creating an identifier for the JAR file called Sum\_JAR:

```
DATABASE JUDF;
CALL SQLJ.INSTALL_JAR('C:\java_udf\sum.jar', 'Sum_JAR', 0);
```

### SQL Definition

The default parameter mapping convention for mapping SQL data types to Java data types is simple mapping, where SQL data types map to Java primitives. In this example, the EXTERNAL NAME string does not specify the parameter types, so simple mapping is assumed. The corresponding Java function should also use simple mapping as shown in the subsequent Java Method Implementation section.

```
DATABASE JUDF;
REPLACE FUNCTION MYSUM(x INTEGER)
  RETURNS INTEGER
  LANGUAGE JAVA
  NO SQL
  CLASS AGGREGATE(200)
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'Sum_JAR:UDFExample.simpleSum';
```

For object mapping, the EXTERNAL NAME string must specify the phase and context along with the object type. For example:

```
DATABASE JUDF;
REPLACE FUNCTION MYSUM(x INTEGER)
  RETURNS INTEGER
  LANGUAGE JAVA
  NO SQL
  CLASS AGGREGATE(200)
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'Sum_JAR:UDFExample.simpleSum(com.teradata.fnc.Phase,
com.teradata.fnc.Context[],java.lang.Integer) returns java.lang.Integer';
```

The corresponding Java UDF definition is:

```
public static java.lang.Integer simpleSum(com.teradata.fnc.Phase phase
,com.teradata.fnc.Context[] context,java.lang.Integer x)
{
    ...
}
```

For details about simple mapping and object mapping, see [Java Data Types](#).

## Example Query

```
SELECT MYSUM(Invoice)
FROM AcctsRec
WHERE (CURRENT_DATE - InvoiceDate) >= 30;
```

## Java Method Implementation

This is an example of a Java aggregate UDF with simple mapping corresponding to the previous SQL function definition with simple mapping.

```
import com.teradata.fnc.*;
import java.io.*;
import java.sql.*;

class Storage implements Serializable{
    double total;

    public Storage() { total = 0.0; }
}

public class UDFExample {

    public static int simpleSum(Phase phase, Context[] context, int x)
    throws SQLException
    {
        try{
            Storage s1 = null;
            /* AGR_DETAIL, AGR_COMBINE, and AGR_FINAL phases
            use the value from the aggregate storage area. */
            if(phase.getPhase()>Phase.AGR_INIT &&
            phase.getPhase()<Phase.AGR_NODATA){

                s1 = (Storage)context[0].getObject(1);
            }
        }
```



```

switch (phase.getPhase()) {

    /* The AGR_INIT phase is executed once. */
    case Phase.AGR_INIT:
        s1 = new Storage();
        context[0].initCtx(s1);
        /* Fall through to detail phase also. */

    case Phase.AGR_DETAIL:
        /* Perform SUM */
        s1.total += x;
        break;

    case Phase.AGR_COMBINE:
        /* SUM between AMPs. */
        Storage s2 = (Storage)context[0].getObject(2);
        s1.total += s2.total;
        break;

    case Phase.AGR_FINAL:
        /* Final, return SUM. */
        return s1.total;

    case Phase.AGR_NODATA:
        /* Not expecting no data. */
        return -1;

    default:
        throw new SQLException("Invalid Phase", "38U05");
}

/* Save the intermediate SUM in the aggregate storage. */
context[0].setObject( 1, s1 );
}catch(Exception ex){
    throw new SQLException(ex.getMessage(),"38101");
}
return -1;
}
}

```

## Java Aggregate Function: Standard Deviation

This example shows the Java code for a simple aggregate UDF that calculates the standard deviation, as follows:

$$S = \sqrt{\frac{\sum X^2}{N} - \left(\frac{\sum X}{N}\right)^2}$$

The Java class defines two methods that calculate the standard deviation. One method shows how to represent intermediate aggregate results as a byte array and the other method shows how to represent intermediate aggregate results as an object.

## JAR File Registration

The following statements register the JAR file for the aggregate UDFs with the JUDF database, creating an identifier for the JAR file called StdDev\_JAR:

```
DATABASE JUDF;
CALL SQLJ.INSTALL_JAR('CJ!java_udf/std_dev.jar', 'StdDev_JAR', 0);
```

## SQL Definition

Here is an SQL UDF definition for the Java method that represents intermediate aggregate results as a byte array.

```
CREATE FUNCTION STD_DEV_BYTE(x FLOAT)
  RETURNS FLOAT
  CLASS AGGREGATE(24)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'StdDev_JAR:UDFExample.fastSTD_DEV(double)
    returns java.lang.Double';
```

Here is an SQL UDF definition for the Java method that represents intermediate aggregate results as an object.

```
CREATE FUNCTION STD_DEV_OBJ(x FLOAT)
  RETURNS FLOAT
  CLASS AGGREGATE(79)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
```

```
EXTERNAL NAME 'StdDev_JAR:UDFExample.STD_DEV(double)
returns java.lang.Double';
```

## Example Query

Consider the following table definition and data:

```
CREATE TABLE Product_Life (Product_ID INTEGER,
                           Product_class VARCHAR(30), HOURS FLOAT);

INSERT INTO Product_Life VALUES (100, 'Bulbs', 100);
INSERT INTO Product_Life VALUES (100, 'Bulbs', 200);
INSERT INTO Product_Life VALUES (100, 'Bulbs', 300);
```

The following query uses the Java method that represents intermediate aggregate results as an object:

```
SELECT Product_ID, SUM(Hours), STD_DEV_OBJ(Hours)
FROM Product_Life
WHERE Product_Class = 'Bulbs'
GROUP BY Product_ID;
```

The following query uses the Java method that represents intermediate aggregate results as a byte array:

```
SELECT Product_ID, SUM(Hours), STD_DEV_BYTE(Hours)
FROM Product_Life
WHERE Product_Class = 'Bulbs'
GROUP BY Product_ID;
```

## Java Implementation

```
import com.teradata.fnc.*;
import java.io.*;
import java.sql.*;

class agr_storage implements Serializable{
    double count;
    double x_sq;
    double x_sum;

    public agr_storage(double a, double b, double c){
        count =a;
        x_sq =b;
```

```

        x_sum = c;
    }
}

public class UDFExample {
    static boolean debug = true;

    public static Double STD_DEV(Phase phase,
                                Context[] context,
                                double x) throws SQLException
    {
        try {
            agr_storage s1 =null;
            agr_storage s2 =null;

            /* The aggregate storage only hold valid data in the
             * AGR_DETAIL, AGR_COMBINE and AGR_FINAL phases. */
            if(phase.getPhase()>Phase.AGR_INIT&&
               phase.getPhase()<Phase.AGR_NODATA){
                s1 = (agr_storage)context[0].getObject(1);

                /* switch to determine the aggregation phase */
                switch (phase.getPhase())
                {
                    /* This phase is executed once per group and */
                    /* allocates and initializes intermediate */
                    /* aggregate storage */
                    case Phase.AGR_INIT:
                        if (debug) System.err.println("Phase: AGR_INIT");

                        /* Get storage for intermediate aggregate values.*/
                        s1 = new agr_storage(0,0,0);
                        context[0].initCtx(s1);

                        /******
                        /* Fall through to detail phase, because the */
                        /* AGR_INIT phase passes in the first set of */
                        /* values for the group. */
                        /******

                        /* This phase is executed once for each selected */
                        /* row to aggregate. One copy will run on each AMP. */
                        case Phase.AGR_DETAIL:

```

```

        if (debug) System.err.println("Phase: AGR_DETAIL");
        s1.count++;
        s1.x_sq += x*x;
        s1.x_sum += x;
        break;

/* This phase combines the results of ALL */
/* individual AMPs for each group.          */
case Phase.AGR_COMBINE:
    if (debug) System.err.println("Phase: AGR_COMBINE");

    /* This is the only phase where Context.getObject(2) */
    /* has a valid value.                                   */
    s2=(agr_storage)context[0].getObject(2);
    s1.count += s2.count;
    s1.x_sq += s2.x_sq;
    s1.x_sum += s2.x_sum;
    break;

/* This phase returns the final result. */
/* It is called once for each group.      */
case Phase.AGR_FINAL:
    if (debug) System.err.println("Phase: AGR_FINAL");
    double term2 = s1.x_sum/s1.count;
    double variance = s1.x_sq/s1.count - term2*term2;

    /* Adjust for deviations close to zero */
    if (Math.abs(variance) < 1.0e-14)
        variance = 0.0;
    return new Double(Math.sqrt(variance));

case Phase.AGR_NODATA:
    if (debug) System.err.println("Phase: AGR_NODATA");

    /* return null if no data */
    return null;

default:
    /* If it gets here there must be an error      */
    /* because this function does not accept any    */
    /* other phase options                          */
    throw new SQLException("Invalid Phase", "38U05");
}
context[0].setObject( 1, s1 );

```

```

    } catch(IOException ex) {
        ex.printStackTrace();
    } catch(ClassNotFoundException e){
        e.printStackTrace();
    }
    return null;
}

public static Double fastSTD_DEV(Phase phase,
                                Context[] context,
                                double x) throws SQLException
{
    /*The intermediate storage is a byte array of 3 doubles*/
    double count=0;
    double x_sq=0;
    double x_sum=0;

    try {
        ByteBuffer s1 =null;
        ByteBuffer s2 =null;

        /* The aggregate storage only hold valid data in the */
        /* AGR_DETAIL, AGR_COMBINE and AGR_FINAL phases.      */
        if (phase.getPhase() > Phase.AGR_INIT &&
            phase.getPhase() < Phase.AGR_NODATA) {
            s1 = ByteBuffer.wrap(context[0].getBytes(1));
            count = s1.getDouble();
            x_sq = s1.getDouble();
            x_sum = s1.getDouble();
        }

        /* switch to determine the aggregation phase */
        switch (phase.getPhase())
        {

            /* This phase is executed once per group and */
            /* allocates and initializes intermediate     */
            /* aggregate storage                           */
            case Phase.AGR_INIT:
                if (debug) System.err.println("Phase: AGR_INIT");
                /* Get storage for intermediate aggregate values.*/
                context[0].initCtx(24);
                /*******/

```

```

/* Fall through to detail phase, because the */
/* AGR_INIT phase passes in the first set of */
/* values for the group. */
/*****

/* This phase is executed once for each selected */
/* row to aggregate. One copy will run on each AMP. */
case Phase.AGR_DETAIL:
    if (debug) System.err.println("Phase: AGR_DETAIL");
    s1 = ByteBuffer.allocate(24);
    count++;
    x_sq += x*x;
    x_sum += x;
    s1.putDouble(count);
    s1.putDouble(x_sq);
    s1.putDouble(x_sum);
    break;

/* This phase combines the results of ALL */
/* individual AMPs for each group. */
case Phase.AGR_COMBINE:
    if (debug) System.err.println("Phase: AGR_COMBINE");
    /*This is the only phase where Context.getBytes(2) */
    /* has a valid value. */
    s2 = ByteBuffer.wrap(context[0].getBytes(2));
    count += s2.getDouble();
    x_sq += s2.getDouble();
    x_sum += s2.getDouble();
    break;

/* This phase returns the final result. */
/* It is called once for each group. */
case Phase.AGR_FINAL:
{
    if (debug) System.err.println("Phase: AGR_FINAL");
    double term2 = x_sum/count;
    double variance = x_sq/count - term2*term2;

    /* Adjust for deviations close to zero */
    if (Math.abs(variance) < 1.0e-14)
        variance = 0.0;
    return new Double(Math.sqrt(variance));
}
case Phase.AGR_NODATA:

```

```

        if (debug) System.err.println("Phase: AGR_NODATA");
        /* return null if no data */
        return null;
    default:
        /* If it gets here there must be an error */
        /* because this function does not accept any */
        /* other phase options */
        throw new SQLException("Invalid Phase", "38U05");
    }
    context[0].setBytes( 1, s1.array() );
} catch(IOException ex){
    ex.printStackTrace();
} catch(ClassNotFoundException e){
    e.printStackTrace();
}
return null; /*will be ignored*/
}
}

```

## Java Window Aggregate Function

This example defines a Java window aggregate function and invokes it using a partition window.

### SQL Definition

```

REPLACE FUNCTION mySum(x INTEGER)
RETURNS INTEGER
CLASS AGGREGATE(1000)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'JavaUDOA:MySumClass.mySum(com.teradata.fnc.Phase,
        com.teradata.fnc.Context[],integer) returns int';

```

### Example Query

The queries in this example reference the following table definition and data:

```

CREATE TABLE t (id INTEGER, v INTEGER);
INSERT INTO t VALUES (1,1);
INSERT INTO t VALUES (1,2);
INSERT INTO t VALUES (1,3);

```



```

INSERT INTO t VALUES (1,4);
INSERT INTO t VALUES (1,5);
INSERT INTO t VALUES (1,6);
INSERT INTO t VALUES (1,7);
INSERT INTO t VALUES (1,8);

SELECT id, v, MYSUM(v) OVER
  (PARTITION BY id ORDER BY v) FROM t;

SELECT id, v, MYSUM(v) OVER
  (PARTITION BY id ORDER BY v ROWS UNBOUNDED PRECEDING) FROM t;

SELECT id, v, MYSUM(v) OVER
  (PARTITION BY id ORDER BY v ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING)
FROM t;

SELECT id, v, MYSUM(v) OVER() FROM t;

```

## Java Implementation

```

import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import com.teradata.fnc.Context;
import com.teradata.fnc.Phase;
import com.ncr.teradata.fnc.*;
import java.io.Serializable;

class Element
{
    public int v;
}

class AGR_Storage implements Serializable
{
    public long current_value; // Current moving sum total
    public int next; // Location to store next detail row
    public int tptr; // Trailing buffer pointer
    public int count; // Number of detail rows processed
    public long window_size; // Moving window size
    public List<Element> data;
    public int heap; // Heap area to store window rows
}

```

```

public AGR_Storage(int a, int b, int c, int d, long e, int f)
{
    current_value = a;
    next = b;
    tptr = c;
    count = d;
    window_size = e;
    data = new ArrayList<Element>(100);
    heap = f;
}
}

public class MySumClass
{
    final private static boolean debug = true;
    public static int MySum(Phase phase, Context context[], int x)
    {
        int result = 0;
        try
        {
            AGR_Storage s1 = null;
            AGR_Storage s2 = null;
            int i, t_x;

            if (phase.getPhase() != Phase.AGR_INIT && phase.getPhase() != Phase.AGR_NODATA)
                s1 = (AGR_Storage) context[0].getObject(1);

            switch (phase.getPhase())
            {
                case Phase.AGR_INIT:
                    if (debug)
                    {
                        System.err.println("Phase: AGR_INIT");
                    }
                    s1 = new AGR_Storage(0,0,0,0,0,0);

                    context[0].initCtx(s1);
                    s1.window_size = context[0].getWindowSize();
                    if(s1.window_size > 0)
                    {
                        for (i=0; i < s1.window_size; i++) //initialize heap
                        {
                            s1.data.get(i).v = 0;
                        }
                    }
                }
            }
        }
        catch (Exception e)
        {
            // Handle exception
        }
    }
}

```

```

    }
}
// fall through

case Phase.AGR_DETAIL:
    t_x = x;
    if (s1.window_size < 0) // non moving case
    {
        s1.current_value += t_x;
    }
    else // Moving
    {
        s1.count++;
        if (s1.count > s1.window_size) // Remove value row from window
        {
            s1.current_value -= s1.data.get(s1.tptr).v;
            s1.tptr = (int)inc(s1.tptr, (int) s1.window_size);
        }
        s1.current_value += t_x; // Add row to window
        s1.data.get(s1.next).v = t_x;
        s1.next = (int) inc(s1.next, (int) s1.window_size);
    }

    break;

case Phase.AGR_COMBINE:

case Phase.AGR_FINAL:
    result = (int) s1.current_value;
    return result;

case Phase.AGR_MOVINGTRAIL:
    s1.current_value -= s1.data.get(s1.tptr).v;
    s1.tptr = (int) inc(s1.tptr, (int)s1.window_size);
    break;

case Phase.AGR_NODATA:
    if (debug)
        System.err.println("Phase: AGR_NODATA");
    return 0;

default:

    throw new SQLException("Invalid Phase", "U0005");

```

```

        }

        context[0].setObject(1, s1);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return result;
}

private static long inc(int next, int size)
{
    return (next + 1) % size;
}
}

```

## Java Constant/Variable Mode Table Function

This example shows the Java code for a table function that supports both Tbl.TBL\_MODE\_CONST and the Tbl.TBL\_MODE\_VARY modes. The Tbl.TBL\_MODE\_CONST is very simple in that any one AMP will extract the data out of the text string that is passed in.

This example extracts some data out of a 'raw' text field to generate a bunch of rows. The raw data is in this format:

```
store_number,entries:entry[;...]
```

### ***entry***

```
customer_ID,item_ID
```

### ***store\_number***

Number that identifies the store that sold the items to the customers.

### ***entries***

Number of items sold.

***customer\_ID***

Customer identifier.

***item\_ID***

Item identifier.

## JAR File Registration

The following statements register the JAR file for the table UDF with the JUDF database, creating an identifier for the JAR file called UDF\_JAR:

```
DATABASE JUDF;
CALL SQLJ.INSTALL_JAR('CJ!java_udf/udf.jar', 'UDF_JAR', 0);
```

## SQL Definition

The default parameter mapping convention for mapping SQL data types to Java data types is simple mapping, where SQL data types map to Java primitives. In this example, the EXTERNAL NAME string does not specify the parameter types, so simple mapping is assumed. The corresponding Java function should also use simple mapping as shown in the subsequent Java Implementation section.

```
DATABASE JUDF;
CREATE FUNCTION extract_field(Text VARCHAR(32000),
                             From_Store INTEGER)
  RETURNS TABLE (Customer_ID INTEGER,
                  Store_ID INTEGER,
                  Item_ID INTEGER)
  LANGUAGE JAVA
  NO SQL
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.extract_field';
```

This example shows the SQL definition for a table function with object mapping:

```
DATABASE JUDF;
CREATE FUNCTION extract_field(Text VARCHAR(32000),
                             From_Store INTEGER)
  RETURNS TABLE (Customer_ID INTEGER,
                  Store_ID INTEGER,
                  Item_ID INTEGER)
  LANGUAGE JAVA
  NO SQL
```

```

PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.extract_field(java.lang.String,
java.lang.Integer, java.lang.Integer[],
java.lang.Integer[], java.lang.Integer[])';

```

The corresponding Java UDF definition is:

```

public static void extract_field(java.lang.String textstr,
                                java.lang.Integer from_store,
                                java.lang.Integer[] cust_id,
                                java.lang.Integer[] store_id,
                                java.lang.Integer[] item_id)
{
    ...
}

```

For details about simple mapping and object mapping, see [Java Data Types](#).

## Example Query

Here is a sample of how to invoke the table function with constant expression input arguments:

```

SELECT *
FROM TABLE (extract_field('25,2:9005,7896,9004,7839;36,1:737,9387;',
25)) AS t1;

```

Here is a sample of how to invoke the table function using the columns from a derived table as input arguments:

```

SELECT DISTINCT cust.Customer_ID, cust.Item_ID
FROM raw_cust,
     TABLE (extract_field(raw_cust.pending_data, 25))
AS cust
WHERE raw_cust.region = 1;

```

Here is the definition of the raw\_cust table:

```

CREATE SET TABLE raw_cust ,NO FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT
(
     region INTEGER,

```

```
pending_data VARCHAR(32000) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX ( region );
```

Here is a sample of the data in the raw\_cust table:

```
region pending_data
```

```
-----
2 7,2:879,3788,879,4500,390,9004;08,1:500,9056;
1 25,3:9005,3789,9004,4907,398,9004;36,2:738,9387,738,9550;
1 25,2:9005,7896,9004,7839;36,1:737,9387;
```

## Java Implementation

This is an example of a Java table function with simple mapping corresponding to the previous SQL function definition with simple mapping.

```
import com.teradata.fnc.*;
import java.io.*;
import java.sql.*;

/*****
/* The definition of the scratch pad */
*****/

class item implements Serializable
{
    int custid;
    int itemid;
}
class local_ctx implements Serializable{
    int Num_Items;
    int Cur_Item;
    int store_num;
    item[] Item_List;
    public local_ctx() {}
    public local_ctx(int Num_Items,
                     int Cur_Item,
                     int store_num,
                     item[] Item_List)
    {
        this.Num_Items = Num_Items;
        this.Cur_Item = Cur_Item;
        this.store_num = store_num;
```

```

        this.Item_List = Item_List;
    }
}

public class UserDefinedFunctions {

    /**
     * Reset the context block.
     */
    static void Reset(local_ctx info)
    {
        info.Num_Items = 0;
        info.Cur_Item = 0;
        info.Item_List = null;
    }

    /**
     * Extract all of the data now. Actually this routine just
     * takes the items that Prescan built and transfers the data
     * out one item at a time.
     */
    static int Extract(local_ctx info,
                       int[] custid,
                       int[] store,
                       int[] itemid)
    {
        /* check to see if there is something left to extract */
        if (info.Cur_Item == info.Num_Items)
            return 0;

        /* okay let's set the output data only if they want it */
        custid[0] = info.Item_List[info.Cur_Item].custid;
        store[0] = info.store_num;
        itemid[0] = info.Item_List[info.Cur_Item].itemid;

        /* set up for next item the next time */
        info.Cur_Item++;
        return 1;
    }

    /**
     * Do a pre-scan of the text and save the data.
     * The text data that this function processes is in a very
     * simple format:
     */

```



```

/* <storenum>,<num items>:<customer id>,<item number>, ... ; */
/* <storenum>,<num items>: ... */
/*****
static int Prescan(local_ctx info,
                  String Text,
                  int frmstore) throws IOException
{
    int storenum = 0;
    int num_items = 0;

    /* find the data for the store we are interested in */
    String Tscan = Text;
    int startpos = 0;
    while (startpos != (-1)) {
        int numpos = Tscan.indexOf(',',startpos);
        if (numpos == -1) return -1;
        storenum = Integer.parseInt(Tscan.substring(startpos,numpos));
        startpos = numpos+1;
        if(frmstore == storenum) {
            num_items = Integer.parseInt(Tscan.substring(startpos,
                                                         Tscan.indexOf(':',startpos)));
            break;
        }
        if (Tscan.indexOf(';') == -1) return -1;
        else {
            Tscan = Tscan.substring(Tscan.indexOf(';')+1,
                                   Tscan.length());
            startpos = 0;
            if(Tscan.equals("")) return 0;
        }
    }

    if (num_items != 0) {
        info.Item_List = new item[num_items];
    }
    else {
        info.Num_Items = 0;
        return 0;
    }

    /* Now let's find all the entries for the store that we are */
    /* interested in. Skip to first item. */
    if (Tscan.indexOf(':')== -1 || Tscan.indexOf(';')== -1) return -1;
    Tscan = Tscan.substring(Tscan.indexOf(':')+1,Tscan.indexOf(';'));

```

```

int pos = 0;
for (int i=0; i<num_items; i++) {
    info.Item_List[i] = new item();
    int nextcust = Tscan.indexOf(',',pos);
    if(nextcust == -1) return -1;
    info.Item_List[i].custid=
        Integer.parseInt(Tscan.substring(pos,nextcust));
    pos = nextcust + 1;
    int nextitem = Tscan.indexOf(',',pos);
    if (nextitem != -1) {
        info.Item_List[i].itemid =
            Integer.parseInt(Tscan.substring(pos,nextitem));
        pos = nextitem + 1;
    }
    else {
        info.Item_List[i].itemid =
            Integer.parseInt(Tscan.substring(pos,Tscan.length()));
        break;
    }
}

info.Num_Items = num_items;
info.store_num = frmstore;
return num_items;
}

public static void extract_field(String Text, /* field decode */
                                int frmStore, /* data to extract */
                                int[] custid, /* 1st output column for row */
                                int[] store, /* 2nd output column */
                                int[] item)
throws SQLException {

    local_ctx state_info;
    int status;
    int[] phase = new int[1];
    Tbl tbl = new Tbl();

    try{
        /* make sure the function is called in the supported context */
        switch (tbl.getPhase(phase))
        {
            /*****
            /* Process the constant expression case. Only one AMP */

```

```

/* will participate for this example. */
/*****
case Tbl.TBL_MODE_CONST:
/* depending on the phase decide what to do */
switch(phase[0])
{
    case Tbl.TBL_PRE_INIT:
        if(tbl.firstParticipant()){
            /* participant */
            return;
        } else {
            /* don't participate */
            if(!tbl.optout()){
                throw new SQLException("Opt-out failed.",
                                      "38U06");
            }
            return;
        }
    case Tbl.TBL_INIT:
        state_info = new local_ctx();
        /* Get scratch memory to keep track of things */
        Reset(state_info);
        /* Pre-process the Text */
        status = Prescan(state_info, Text, frmStore );
        if (status == -1) {
            throw new SQLException(
                "Text had pre-scan errors","38U08");
        }
        tbl.allocCtx(state_info);
        tbl.setCtxObject(state_info);
        break;
    case Tbl.TBL_BUILD:
        state_info = (local_ctx)tbl.getCtxObject();
        status = Extract(state_info,
                        custid,
                        store,
                        item);

        if (status == 0)
            /* Have no more data, return no data sqlstate */
            throw new SQLException("no more data","02000");
        else if (status == -1){
            throw new SQLException(
                "Text had extract error","38U09");

```

```

    }
    tbl.setCtxObject(state_info);
    break;
case Tbl.TBL_END:
    /* everyone done */
    state_info = (local_ctx)tbl.getCtxObject();
    break;
}
break;
/*****
/* Process the varying expression */
*****/
case Tbl.TBL_MODE_VARY:
    switch(phase[0])
    {
        case Tbl.TBL_PRE_INIT:
            item[] itemarr = new item[3];
            for (int i=0; i < itemarr.length; i++)
                itemarr[i] = new item();
            state_info = new local_ctx(0,0,0,itemarr);
            /* get scratch memory to use from now on */
            tbl.allocCtx(state_info);
            break;
        case Tbl.TBL_INIT:
            state_info = new local_ctx();
            /* Pre-process the Text */
            status = Prescan(state_info, Text, frmStore );
            if (status == -1) {
                if (tbl.abort()) {
                    throw new SQLException(
                        "Text had pre-scan errors","38U08");
                }
            }
            tbl.setCtxObject(state_info);
            break;
        case Tbl.TBL_BUILD:
            state_info = (local_ctx)tbl.getCtxObject();
            status = Extract(state_info,
                            custid,
                            store,
                            item);
            if (status == 0)
                /* Have no more data return no data sqlstate */
                throw new SQLException("no more data","02000");
    }
}

```

```

        else if (status == -1) {
            /* if I was the first then report the error */
            if(tbl.abort()) {
                throw new SQLException(
                    "Text had extract error","38U09");
            }
            return;
        }
        tbl.setCtxObject(state_info);
        break;
    case Tbl.TBL_FINI:
        /* initialize for the next set of data */
        state_info = (local_ctx)tbl.getCtxObject();
        Reset(state_info);
        break;
    case Tbl.TBL_END:
        break;
    case Tbl.TBL_ABORT:
        break;
    }
    return;
}
} catch(ClassNotFoundException e) {
    e.printStackTrace();
} catch(IOException e) {
    e.printStackTrace();
}
}
}

```

## Java Table Operator

This example shows the Java code for a table operator.

## JAR File Registration

These statements register the JAR file for the table operator with the JUDF database, creating an identifier for the JAR file called 'example':

```

DATABASE JUDF;
CALL SQLJ.INSTALL_JAR('CJ!C:\temp\example.jar','example',0);

```

## SQL Definition

```
REPLACE FUNCTION Sessionize()
RETURNS TABLE VARYING USING FUNCTION session_contract
LANGUAGE JAVA NO SQL
PARAMETER STYLE SQLTABLE
EXTERNAL NAME 'example:examples.Sessionize.execute';
```

## Example Query

```
SELECT * FROM Sessionize (ON td2 USING TimeCol('eventinfo') timeout(30))
AS J1;
```

```
CREATE SET TABLE UT1.td2 ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
    (
    guid INTEGER,
    payload VARCHAR(2048) CHARACTER SET LATIN NOT CASESPECIFIC,
    eventinfo TIMESTAMP(6))
;
```

```
SELECT guid, sessionid FROM Sessionize (ON td2 USING
TimeCol('eventinfo') timeout(30)) AS J1;
```

\*\*\* Query completed. 5 rows found.

GUID	SessionID
33	1
33	2
33	2
33	2
33	3

```
SELECT * FROM td2;
```

\*\*\* Query completed. 5 rows found.

guid	payload	eventinfo
------	---------	-----------

```

-----
33 http://dougfy.wikidot.com/debu 2013-06-09 20:21:44.000000
33 http://dougfy.wikidot.com/debu 2013-06-10 19:08:55.000000
33 http://dougfy.wikidot.com/debu 2013-06-10 19:09:01.000000
33 http://dougfy.wikidot.com/debu 2013-06-10 19:09:04.000000
33 http://dougfy.wikidot.com/debu 2013-06-10 19:09:41.000000

```

## Java Method Implementation

```

package examples;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;
import com.teradata.fnc.TeradataType;
import com.teradata.fnc.operator.Metadata;
import com.teradata.fnc.operator.TableOperator;
import com.teradata.fnc.operator.TeradataResultSet;
import com.teradata.fnc.runtime.ColumnDefinition;
import com.teradata.fnc.runtime.RuntimeContract;
import com.teradata.fnc.value.MismatchException;
import com.teradata.fnc.value.RangeException;
import com.teradata.fnc.value.Value;
public class Sessionize implements TableOperator{

    public int contract(RuntimeContract contract, ResultSet[] rsin,
ResultSet[] arg2)
        throws SQLException {

        /* Create the number of output columns plus additional SessionID. */
        Metadata iCols = ((TeradataResultSet)rsin[0]).getTeradataMetaData();
        ColumnDefinition OutCols[] = new ColumnDefinition[iCols.getColumnCount() + 1];
        int col = 0;

        /* Copy input columns to output columns. */
        for (col=0;col<iCols.getColumnCount();col++) {
            OutCols[col] = new
ColumnDefinition(iCols.getColumnName(col+1), iCols.getTeradataColumnType(col+1));
            OutCols[col].setDisplayLength(iCols.getColumnDisplaySize(col+1));
            switch (TeradataType.get(iCols.getTeradataColumnType(col+1))) {
                case VARCHAR_DT:
                case CHAR_DT:
                    OutCols[col].setCharset(iCols.getPrecision(col+1));

```

```

        break;
    case VARBYTE_DT:
        OutCols[col].setCharset(iCols.getPrecision(col+1));
        OutCols[col].setDisplayLength(iCols.getColumnDisplaySize(col+1));
        break;
    case TIME_DT:
    case TIMESTAMP_DT:
    case TIME_WTZ_DT:
    case TIMESTAMP_WTZ_DT:
        OutCols[col].setPrecision(iCols.getPrecision(col+1));
        break;
    default:
        OutCols[col].setPrecision(iCols.getPrecision(col+1));
        OutCols[col].setScale(iCols.getScale(col+1));
        break;
    }
}

/* Add SessionID column. */
OutCols[col] = new ColumnDefinition("SessionID", TeradataType.INTEGER_DT);
OutCols[col++].setDisplayLength(4);

/* Set output columns and complete the contract. */
contract.setOutputInfo(0, OutCols);
contract.complete();
return 1;

} /* contract */

public void execute(RuntimeContract contract, ResultSet[] rsin,
ResultSet[] rsout)
    throws SQLException {

    try {
        int currentSessionId = 0;
        long lastTime = 0;
        String TimeColumn =
(String)((Value)contract.getInputInfo().getCustom().get("TimeCol")).getObject();
        byte timevalue
= ((Value)contract.getInputInfo().getCustom().get("timeout")).getByte();
        long window = timevalue*1000;
        int colcount = rsin[0].getMetaData().getColumnCount();

        /* Copy user data adding a sessionization column. */

```



```

while (rsin[0].next()) {
    /* Determine if time of this click is more than the window after the last. */
    Timestamp currentTime = (Timestamp) rsin[0].getObject(TimeColumn);
    if ( currentTime.getTime() > lastTime + window) {
        currentSessionId++;
    }

    for(int i=1;i<=colcount;i++) {
        Object o = rsin[0].getObject(i);
        if(rsin[0].wasNull())
            rsout[0].updateObject(i, null);
        else {
            rsout[0].updateObject(i, o);
        }
    }

    /* Add session info. */
    rsout[0].updateObject(colcount+1, currentSessionId);
    rsout[0].insertRow();
    lastTime = currentTime.getTime();
}
} catch (MismatchException e1) {
    throw new SQLException("T0002", "Invalid timeout value numeric");
} catch (RangeException e1) {
    throw new SQLException("T0002", "Invalid timeout value 1-30");
}

} /* execute */
}

```

# External Stored Procedure Code Examples

This section provides C and Java code examples of external stored procedures.

## Simple External Stored Procedure

This example shows the C code for an external stored procedure that takes an INOUT string argument, strips off the first four characters, and returns the result.

### SQL Definition

```
CREATE PROCEDURE GetRegionXSP
  (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE TD_GENERAL;
```

### Sample Query

The following query uses the GetRegionXSP external stored procedure to strip off the first four characters of the region INOUT argument and return the result.

```
USING (region VARCHAR(64))
CALL GetRegionXSP(:region);
```

### C Function Definition

```
/****** C source file name: getregion.c *****/

#define SQL_TEXT Latin_Text

#include <sqltypes_td.h>
#include <string.h>
void xsp_getregion( VARCHAR_LATIN *region,
                  char sqlstate[6])
{
    char tmp_string[64];

    if (strlen((const char *)region) > 4)
```

```

{
    /* Strip off the first four characters */
    strcpy(tmp_string, (char *)region);
    strcpy((char *)region, &tmp_string[4]);
}
}

```

## Calling a Stored Procedure Using FNC\_CallSP

This example extends the preceding example, [Simple External Stored Procedure](#), by calling the FNC\_CallSP library function to call the addRegion stored procedure. The external stored procedure strips off the first four characters of the region input argument and passes the result to the addRegion stored procedure, which inserts the string into the regionTable table.

The addRegion stored procedure is defined like this:

```

CREATE PROCEDURE addRegion (IN region VARCHAR(64),
                           OUT region_count INTEGER)
BEGIN
    INSERT INTO regionTable VALUES (:region);
    SEL COUNT(*) INTO :region_count FROM regionTable;
END;

```

The regionTable table is defined like this:

```

CREATE TABLE regionTable (region VARCHAR(64));

```

## SQL Definition of the External Stored Procedure

```

CREATE PROCEDURE GetRegionXSP
    (INOUT region VARCHAR(64))
LANGUAGE C
NO SQL
EXTERNAL NAME 'CS!getregion!xspsrc/getregion.c!F!xsp_getregion'
PARAMETER STYLE SQL;

```

## Sample Query

The following query uses the GetRegionXSP external stored procedure to strip off the first four characters of the region INOUT argument and return the result.

```

USING (region VARCHAR(64))
CALL GetRegionXSP(:region);

```

## C Function Definition

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void xsp_getregion ( VARCHAR_LATIN *region,
                    int           *region_isnull,
                    char          sqlstate[6],
                    SQL_TEXT      extname[129],
                    SQL_TEXT      specific_name[129],
                    SQL_TEXT      error_message[257])
{
    char    tmp_string[64];
    void    *argv[2];
    int     ind[2];
    parm_t  dtype[2];
    INTEGER regionCount; /* OUT argument from addRegion */

    /* Set the return indicator value for the external stored procedure*/
    *region_isnull = 0;

    if (strlen((const char *)region) > 4)
    {
        /* Strip off the first four characters */
        strcpy(tmp_string, (char *)region);
        strcpy((char *)region, &tmp_string[4]);

        /* Set the pointers to the stored procedure arguments */
        argv[0] = region;          /* IN */
        argv[1] = &regionCount;    /* OUT */

        /* Set the indicator for the IN argument */
        ind[0] = 0;

        memset(dtype, 2, sizeof(parm_t)*2);

        /* Data type for the VARCHAR IN argument */
        dtype[0].datatype = VARCHAR_DT;
    }
}

```

```

dtype[0].direction = IN_PM;
dtype[0].charset = LATIN_CT;
dtype[0].size.length = strlen((const char *)region);

/* Data type for the INTEGER OUT argument */
dtype[1].datatype = INTEGER_DT;
dtype[1].direction = OUT_PM;

FNC_CallSP((SQL_TEXT *)"addRegion", 2, argv, ind, dtype, sqlstate);

if (strcmp(sqlstate, "00000") != 0)
{
    strcpy((char *)error_message, "Bad call to addRegion");
    return;
}
else
{
    strcpy(sqlstate, "U00001");
    strcpy((char *)error_message, "Region string too short");
    return;
}
}

```

## Sending Mail

This example shows how to send mail on Linux by calling sendmail in an external stored procedure. It is very primitive and does no error checking as good programming practices should. But it does show how relatively easy it is to get an external stored procedure to do something useful. The procedure can be called from within a trigger action body if desired.

Because the external stored procedure does I/O, the definition includes the EXTERNAL SECURITY clause.

## SQL Definition

```

CREATE AUTHORIZATION mail_token
AS DEFINER DEFAULT
  USER 'mailmgr'
  PASSWORD 'secret';

REPLACE PROCEDURE tdmmail(in From_ VARCHAR(100),
                          in To_ VARCHAR(100),
                          in Subject VARCHAR(200),
                          in Message VARCHAR(32000))

```

```

LANGUAGE C
NO SQL
PARAMETER STYLE SQL
EXTERNAL name 'CS!tdmail!tdmail.c'
EXTERNAL SECURITY DEFINER;

```

## Sample Query

The following query calls the external stored procedure:

```

CALL tdmail(NULL,
             'somebody@teradata.com',
             'Just to say hello',
             'Today is a good day to get some rest.');
```

## C Function Definition

```

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <stdio.h>
#include <stdlib.h>

/*****
/* exec sendmail to send the mail. It returns -1 if it failed.  */
*****/

static int send_to_system(char *mail_msg, int from_addr)

{
    char c;
    char *mailcmd;

    FILE *mailhndl;

    /*****
    /* sets up the send mail command */
    *****/

    /* If the caller did not provide a from user then supply "mailmgr" */
    if (from_addr == -1)
        mailcmd = "/usr/sbin/sendmail -oi -f mailmgr -oem -t";
    else
        mailcmd = "/usr/sbin/sendmail -oi -oem -t";

```

```

/* Execute the send mail command */
mailhdl = (FILE *) popen(mailcmd, "w");

/* If the handle is not null then at least it took it */
if (mailhdl == NULL)
    return -1;

/* Pipe the message otherwise there is not going to be any mail */
while ((c= *mail_msg++) != 0)
    putc(c,mailhdl);

/* sent the whole message- close the pipe to let it know we are done */
pclose(mailhdl);

return 0;
}

/*****
/* The External Stored Procedure */
*****/

void tdmail( VARCHAR_LATIN *From,      /* who it is from */
             VARCHAR_LATIN *To,        /* For who Name@domain from */
             VARCHAR_LATIN *Subject,   /* Need a subject */
             VARCHAR_LATIN *Message,   /* What you want to say */
             int             *i_from,
             int             *i_to,
             int             *i_subject,
             int             *i_message,
             char             sqlstate[6],
             SQL_TEXT        extname[129],
             SQL_TEXT        specific_name[129],
             SQL_TEXT        error_message[257])
{
    char *mailbuf;
    int status;
    char *from_usr;
    char *subject_usr;
    char *message_usr;

    /* Format the email into standard form for mail. */

```

```

/* If there is no "TO" address then it is not going anywhere. */
/* There might be other problems. We probably should check for a */
/* properly formatted sender address of the form <name>@<domain>. */
if (*i_to == -1)
{
    strcpy((char *) sqlstate, "U0001");
    strcpy((char *) error_message, "There is no 'To' address");
    return;
}

/* If no "FROM" user, then use mailmgr. */
if (*i_from == -1)
    from_usr = "mailmgr";
else
    from_usr = (char *) From;

/* No subject. Just substitute or make one up. */
if (*i_subject == -1)
    subject_usr = "<no subject>";
else
    subject_usr = (char *) Subject;

/* No message. Just make one up. */
if (*i_message == -1)
    message_usr = "<Empty Message>";
else
    message_usr = (char *) Message;

/* Need to allocate memory to hold formatted message. */
mailbuf = FNC_malloc(strlen(From)+
                    strlen(To) +
                    strlen(Subject)+
                    strlen(Message)+ 1000);

/* This will format it. The assumption is that the message */
/* part is already formatted with returns and the like. */
sprintf(mailbuf,
        "From: %s\nTo: %s\nSubject: %s\n\n%s\n\0",
        from_usr,
        To,
        subject_usr,
        message_usr);

```



```

/* Call sendmail to send it off. */
status = send_to_system(mailbuf, *i_from);
FNC_free(mailbuf);

/* Well we got an error back. At this point just send a warning. */
if (status < 0)
{
    strcpy(sqlstate, "01H001");
    strcpy((char *) error_message, "Error: mail not sent");
}
}

```

## Using CLlv2 to Execute SQL

This example shows how to use CLlv2 in an external stored procedure to directly execute the following SQL requests:

- DELETE USER
- DROP USER
- DROP ROLE
- CREATE ROLE
- GRANT
- DROP PROFILE
- CREATE PROFILE
- CREATE USER

The external stored procedure uses the first input argument as the name for the CREATE USER request.

## SQL Definition

```

REPLACE PROCEDURE CliXSP_ETUser.ET001_xsp1(IN A_Name VARCHAR(10),
                                           OUT resultx VARCHAR(16000))

LANGUAGE C
MODIFIES SQL DATA
PARAMETER STYLE SQL
EXTERNAL NAME 'SP!CLI!CS!ET001_xsp1!ET001_xsp.c';

```

## Sample Query

The following query calls the external stored procedure to create a new user with a name of u001:

```
CALL CliXSP_ETUser.ET001_xsp1('u001', resultx);
```

## C Function Definition

```

/***** C source file name: ET001_xsp.c *****/

#define SQL_TEXT Latin_Text

#include <sqltypes_td.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <coptypes.h>
#include <coperr.h>
#include <parcel.h>
#include <dbcarea.h>
#include <dbchqep.h>

#define CONNECTED 0
#define NOT_CONNECTED 1
#define OK 0
#define STOP 1
#define FAILED -1
#define MAX_SESSIONS 1

#define D8CAIRXSIZ 24
#define FIXED_ELM_LEN_8 42
#define D8XILMNTSIZE 4
#define SPOPTIONSSIZE 2

struct ERROR_FAIL_Type
{
    Word StatementNo;
    Word Info;
    Word Code;
    Word Length;
    char Msg[255];
} ;

/*****
    dbc_init() Initialize DBCAREA.
*****/
static Int16

```

```

dbc_init(DBCAREA *dbcarea_pt,
        SQL_TEXT *error_message)
{
    Int32 result;
    char cnta[4];

    dbcarea_pt->total_len = sizeof(struct DBCAREA);
    DBCHINI(&result, cnta, dbcarea_pt);
    if (result != EM_OK)
    {
        sprintf((char *) error_message,
                "Error in initialization --%s", dbcarea_pt->msg_text);
        return result;
    }
    return EM_OK;
}

/*****
    set_options() Set DBCAREA options.
*****/
static void
set_options(DBCAREA *dbcarea_pt)
{
    dbcarea_pt->change_opts          = 'Y';
    dbcarea_pt->use_presence_bits    = 'N';
    dbcarea_pt->keep_resp            = 'N';
    dbcarea_pt->loc_mode              = 'Y';
    dbcarea_pt->var_len_req           = 'N';
    dbcarea_pt->save_resp_buf         = 'N';
    dbcarea_pt->two_resp_bufs         = 'N';
    dbcarea_pt->ret_time              = 'N';
    dbcarea_pt->wait_for_resp         = 'Y';
    dbcarea_pt->req_proc_opt          = 'E';
    dbcarea_pt->req_buf_len           = 1024;
    dbcarea_pt->resp_buf_len          = 1024;
    dbcarea_pt->data_encryption        = 'N';
    dbcarea_pt->create_default_connection = 'Y';
}

/*****
    dbc_con() Establish a default connection.
*****/
static Int16
dbc_con(DBCAREA *dbcarea_pt,

```

```

        SQL_TEXT *error_message)
{
    char cnta[4];
    Int32 result;

    dbcarearea_pt->func=DBFCON;
    DBCHCL(&result, cnta, dbcarearea_pt);
    if (result != EM_OK)
    {
        sprintf((char *)error_message,
                "Error in Logon--%s\n", dbcarearea_pt->msg_text);
        return result;
    }
    return EM_OK;
}

/*****
    dbc_irq()  Initiate request.
*****/
static Int16
dbc_irq(DBCAREA *dbcarearea_pt,
        SQL_TEXT *error_message,
        char *str)
{
    Int32 result;
    char cnta[4];
    dbcarearea_pt->func = DBFIRQ;
    dbcarearea_pt->req_ptr = str;
    dbcarearea_pt->req_len = strlen(str);
    DBCHCL(&result, cnta, dbcarearea_pt);
    if (result != EM_OK)
    {
        sprintf((char *)error_message,
                "Error in initiating a request--%s",
                dbcarearea_pt->msg_text);
        return result;
    }
    return EM_OK;
}

/*****
    fetch_request()  Fetch parcels and check activity type.
*****/

```

```

static Int16
fetch_request(DBCAREA *dbcarea_pt,
              int stmt_no,
              char *resultx,
              SQL_TEXT *error_message)
{
    long request, session;

    Int32 result;
    char cnta[4];
    union
    {
        int  acc_value;
        unsigned char  acc_c[4];
    } acc;
    char strptr[256];
    int count, i, status, rowcount;
    int ShowFlag = 0;
    char ctc[6400];
    struct CliSuccessType *SuccPcl;
    struct CliOkType *OKPcl;
    struct CliFailureType *FailPcl;

    request = dbcarea_pt->o_req_id;
    session = dbcarea_pt->o_sess_id;

    dbcarea_pt->i_sess_id = session;
    dbcarea_pt->i_req_id = request;
    dbcarea_pt->func = DBFFET;
    status=OK;
    rowcount=1;
    while (status == OK)
    {
        DBCHCL(&result, cnta, dbcarea_pt);
        count=1;
        if (result == REQEXHAUST) status = STOP;
        else if (result != EM_OK) status = FAILED;
        else
        {
            switch(dbcarea_pt->fet_parcel_flavor)
            {
                case PclSUCCESS :
                    SuccPcl =

```

```

        (struct CliSuccessType *)dbcarea_pt->fet_data_ptr;
    memcpy(acc.acc_c, SuccPcl->ActivityCount, 4);
    sprintf(strptr, "[%d]Succ:Act(%d)|\0",
        stmt_no, SuccPcl->ActivityType);
    strcat(resultx, strptr);
    break;
case PclOK :
    OKPcl = (struct CliOkType *)dbcarea_pt->fet_data_ptr;
    sprintf(strptr, "[%d]OK:Act(%d)|\0",
        stmt_no, OKPcl->ActivityType);
    strcat(resultx, strptr);
    break;
case PclRECORD :
    memcpy(&ctc[0], &dbcarea_pt->fet_data_ptr[0],1);
    memcpy(&ctc[1], &dbcarea_pt->fet_data_ptr[1],1);
    memcpy(&ctc[2], &dbcarea_pt->fet_data_ptr[2],1);
    break;
case PclENDSTATEMENT :
    break;
case PclFAILURE :
    FailPcl =
        (struct CliFailureType *)dbcarea_pt->fet_data_ptr;
    sprintf(strptr, "[%d]Fail:Cde(%d)|\0",
        stmt_no, FailPcl->Code);
    strcat(resultx, strptr);
    break;
case PclERROR :
    sprintf(strptr, "[%d]PclError");
    strcat(resultx, strptr);
    status = STOP;
    return -1;
} /* end of switch */
} /* end of else */
} /* end of while */
if (status == FAILED)
{
    sprintf((char *)error_message,
        "Error in fetching a request--%s",dbcarea_pt->msg_text);
    return status;
}
return EM_OK;
}

/*****

```

```

    end_request() End request.
    *****/
static Int16
end_request(DBCAREA *dbcarea_pt,
            SQL_TEXT *error_message)
{
    Int32 result;
    char cnta[4];

    dbcarea_pt->i_sess_id = dbcarea_pt->o_sess_id;
    dbcarea_pt->i_req_id = dbcarea_pt->o_req_id;
    dbcarea_pt->func = DBFERQ;
    DBCHCL(&result, cnta, dbcarea_pt);

    if (result != EM_OK)
    {
        sprintf((char *)error_message,
                "Error in EndRequest--%s", dbcarea_pt->msg_text);
        return result;
    }
    return EM_OK;
}

/*****/
Entry point for external stored procedure.
*****/
void
ET001_xsp1(VARCHAR_LATIN *A_Name,
            VARCHAR_LATIN *result,
            int *a_name_i,
            int *result_i,
            char sqlstate[6],
            SQL_TEXT extname[129],
            SQL_TEXT specific_name[129],
            SQL_TEXT error_message[257])
{
    DBCAREA dbcarea;

    char str1[200]; /* DELETE USER request */
    char str2[200]; /* DROP USER request */
    char str3[200]; /* DROP ROLE R_001 request */
    char str4[200]; /* CREATE ROLE R_001 request */
    char str5[200]; /* GRANT ... to R_001 request */

```

```

char str6[200]; /* DROP PROFILE request */
char str7[200]; /* CREATE PROFILE P_001 request */
char str8[200]; /* CREATE USER request */

if (dbc_init(&dbcarea, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0005");
    return;
}
set_options(&dbcarea);
if (dbc_con(&dbcarea, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0006");
    return;
}
if (fetch_request(&dbcarea, 0, (char *) result, error_message)
    != EM_OK)
{
    strcpy (sqlstate, "U0007");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0008");
    return;
}

/***** DELETE USER request *****/
memset(&str1, ' ', 100);
sprintf(str1, "Delete User %s;", A_Name);
if (dbc_irq(&dbcarea, error_message, str1) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 1, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

```



```

}

/***** DROP USER request *****/
memset(&str2, ' ', 100);
sprintf(str2, "Drop User %s;", A_Name);
if (dbc_irq(&dbcarea, error_message, str2) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 2, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

/***** DROP ROLE request *****/
memset(&str3, ' ', 100);
sprintf(str3, "Drop Role R_001;");
if (dbc_irq(&dbcarea, error_message, str3) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 3, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

/***** CREATE ROLE request *****/
memset(&str4, ' ', 100);
sprintf(str4, "CREATE ROLE R_001;");
if (dbc_irq(&dbcarea, error_message, str4) != EM_OK)

```

```

{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 4, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

/***** GRANT request *****/
memset(&str5, ' ', 100);
sprintf(str5,
    "GRANT ALL BUT CREATE USER, DROP USER ON CliXSP_ETUser TO R_001;");
if (dbc_irq(&dbcarea, error_message, str5) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 5, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

/***** DROP PROFILE request *****/
memset(&str6, ' ', 100);
sprintf(str6, "DROP PROFILE P_001;");
if (dbc_irq(&dbcarea, error_message, str6) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 6, (char*)result, error_message) != EM_OK)

```

```

{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

/***** CREATE PROFILE request *****/
memset(&str7, ' ', 100);
sprintf(str7, "CREATE PROFILE P_001 AS ACCOUNT = '&M', SPOOL = 100000,
TEMPORARY = 100000;");
if (dbc_irq(&dbcarea, error_message, str7) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 7, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}
if (end_request(&dbcarea, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}

/***** CREATE USER request *****/
memset(&str8, ' ', 100);
sprintf(str8, "CREATE USER %s from CliXSP_ETUser AS PERM = 100000,
PASSWORD = u001, DEFAULT ROLE = R_001, PROFILE = P_001;", A_Name);
if (dbc_irq(&dbcarea, error_message, str8) != EM_OK)
{
    strcpy(sqlstate, "U0009");
    return;
}
if (fetch_request(&dbcarea, 8, (char*)result, error_message) != EM_OK)
{
    strcpy (sqlstate, "U0010");
    return;
}

```

```

if (end_request(&dbcare, error_message) != EM_OK)
{
    strcpy(sqlstate, "U0011");
    return;
}
}

```

## Using CLlv2 to Consume Dynamic Result Sets

This example uses CLlv2 in an external stored procedure to execute an SQL request to call a stored procedure that returns dynamic result sets.

The external stored procedure does the following:

- Drops and creates a table called TraceTable that it uses to trace the types of parcels that are returned to the external stored procedure as a result of calling the stored procedure.
- Drops and creates a table called tab1.
- Inserts 10 rows into tab1.
- Calls the p1 stored procedure, which selects the 10 rows from the tab1 table. For each parcel returned, the external stored procedure inserts that into the TraceTable table.

The stored procedure that returns the result sets is defined like this:

```

REPLACE PROCEDURE P1 (IN OUT_SQLSTR VARCHAR(50))
DYNAMIC RESULT SETS 1
BEGIN
    DECLARE C1 CURSOR WITH RETURN ONLY TO CALLER FOR S1;
    PREPARE S1 FROM OUT_SQLSTR;
    OPEN C1;
END;

```

For details on the REPLACE PROCEDURE statement and the DYNAMIC RESULT SETS clause, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144. For details on the DECLARE CURSOR, PREPARE, and OPEN statements, see *Teradata Vantage™ - SQL Stored Procedures and Embedded SQL*, B035-1148.

## SQL Definition for the External Stored Procedure

```

REPLACE PROCEDURE sqlxspex()
LANGUAGE C
MODIFIES SQL DATA
PARAMETER STYLE SQL
EXTERNAL NAME 'SP!CLI!CS!sqlxspex!sqlxsptests/sqlxspexample.c';

```

## Sample Query

The following query calls the external stored procedure:

```
CALL sqlxspex();
```

To see the data that the external stored procedure stores on the parcels that it receives after calling the stored procedure, use the following statement:

```
SELECT * FROM TraceTable ORDER BY 1;
```

## C Function Definition

```

/***** C source file name: sqlxspexample.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <coptypes.h>
#include <coperr.h>
#include <parcel.h>
#include <dbcarea.h>
#include <dbchqep.h>

typedef struct
{
    Word          StatementNo;
    Word          Info;
    Word          Code;
    Word          Length;
    char          Msg[1];          /* 1 to 255 bytes */
} PclErrorFailBody, *PclErrorFailBody_pt;

struct DBCAREA DBCArea;
char * pContext;
DBCAREA *pDBCArea;
DBCAREAX *pDbcAreaX;

/*****
    Initialize()  Initialize DBCAREA.
*****/

```

```

*****/
Int32 Initialize()
{
    Int32 Result;

    pDBCArea = &DBCAREA;
    pDBCArea->total_len = sizeof(struct DBCAREA);

    DBCHINI(&Result, pContext, &DBCAREA);

    pDBCArea->change_opts      = 'Y' ;
    pDBCArea->req_buf_len      = 1024;
    pDBCArea->resp_buf_len     = 1024;
    pDBCArea->use_presence_bits = 'N' ;
    pDBCArea->keep_resp        = 'N' ;
    pDBCArea->loc_mode         = 'Y' ;
    pDBCArea->var_len_req       = 'N' ;
    pDBCArea->save_resp_buf    = 'N' ;
    pDBCArea->two_resp_bufs    = 'N' ;
    pDBCArea->ret_time         = 'N' ;
    pDBCArea->wait_for_resp     = 'Y' ;
    pDBCArea->req_proc_opt     = 'E' ;
    pDBCArea->maximum_parcel    = 'H';
    pDBCArea->resp_mode        = 'I';
    pDBCArea->dynamic_result_sets_allowed = 'Y';

    return Result;
}

/*****
    Logon() Establish a default connection.
*****/
Int32 Logon(char *LogonStr)
{
    Int32 Result;
    pDBCArea->func=DBFCON;
    if (LogonStr == NULL)
    {
        pDBCArea->create_default_connection = 'Y';
    }
    else
    {
        pDBCArea->logon_len = (Int32) strlen(LogonStr);
        pDBCArea->logon_ptr = LogonStr;
    }
}

```

```

    }

    DBCHCL(&Result, pContext, pDBCArea);
    if (Result != EM_OK)
        return Result;

    /* Fetch the response */
    pDBCArea->i_sess_id = pDBCArea->o_sess_id;
    pDBCArea->i_req_id  = pDBCArea->o_req_id;
    pDBCArea->func      = DBFFET;
    DBCHCL(&Result, pContext, pDBCArea);
    if (Result != EM_OK)
        return Result;

    if ( pDBCArea->fet_parcel_flavor != PclSUCCESS )
    {
        if (    pDBCArea->fet_parcel_flavor == PclFAILURE
            || pDBCArea->fet_parcel_flavor == PclERROR )
        {
            return pDBCArea->fet_parcel_flavor;
        }
    }

    /* End the logon request */
    pDBCArea->i_sess_id = pDBCArea->o_sess_id;
    pDBCArea->i_req_id  = pDBCArea->o_req_id;
    pDBCArea->func      = DBFERQ;
    DBCHCL(&Result, pContext, pDBCArea);
    return Result;
}

/*****
    Logoff() Disconnect.
*****/
Int32 Logoff()
{
    Int32 Result;

    pDBCArea->func = DBFDSC;
    DBCHCL(&Result, pContext, pDBCArea);
    return Result;
}

/*****

```

```

    Execute() Execute the specified request.
    *****/
Int32 Execute(char *ReqStr)
{
    Int32 Result;

    /* Issue the request */
    pDBCArea->func      = DBFIRQ;
    pDBCArea->change_opts = 'Y';
    pDBCArea->req_ptr    = ReqStr;
    pDBCArea->req_len    = (Int32) strlen(pDBCArea->req_ptr);
    DBCHCL(&Result, pContext, pDBCArea);
    if (Result != EM_OK) return Result;

    /* Fetch the result parcels */
    pDBCArea->func = DBFFET;
    pDBCArea->change_opts = 'Y';
    pDBCArea->i_req_id = pDBCArea->o_req_id;

    while (Result == EM_OK)
        DBCHCL(&Result, pContext, pDBCArea);

    /* End the request */
    pDBCArea->func = DBFERQ;
    DBCHCL(&Result, pContext, pDBCArea);
    return Result;
}

/*****
    DoTheWork() Create the tables, call the stored procedure.
    *****/
void DoTheWork()
{
    Int32      Result;
    Int32      a;
    Int32      CallReqNo;
    Int32      Flavor;
    char       ReqStr[255];
    Int32*     pInt32;
    char*      pChar;

    /* Create a trace table */
    Result = Execute("DROP TABLE TraceTable;");
    if (Result != EM_OK) return;

```



```

    Result = Execute("CREATE TABLE TraceTable(SeqNo INT GENERATED BY
DEFAULT AS IDENTITY, Flavor INT, Text VARCHAR(100));");
    if (Result != EM_OK) return;

    /* Create a table to insert and select rows */
    Result = Execute("DROP TABLE tab1;");
    if (Result != EM_OK) return;

    Result = Execute("CREATE TABLE tab1(a INT, b INT);");
    if (Result != EM_OK) return;

    /* Insert the rows */
    for (a=1;a<=10;a++)
    {
        pDBCArea->using_data_ptr      = (char *) &a;
        pDBCArea->using_data_len      = sizeof(a);
        Result = Execute("USING(a INT) INSERT tab1(:a, :a);");
        if (Result != EM_OK) return;
    }

    /* Call a procedure which will return a result set */
    pDBCArea->func      = DBFIRQ;
    pDBCArea->change_opts = 'Y';
    pDBCArea->req_ptr    = "CALL p1('SEL * FROM tab1 ORDER BY 1')";
    pDBCArea->req_len    = (Int32) strlen(pDBCArea->req_ptr);
    DBCHCL(&Result, pContext, pDBCArea);
    if (Result != EM_OK) return;

    /* Remember the request number */
    CallReqNo = pDBCArea->o_req_id;

    /* Fetch the result parcels and insert into the trace table */
    while (Result == EM_OK)
    {
        /* Fetch a parcel from the CALL */
        pDBCArea->func = DBFFET;
        pDBCArea->change_opts = 'Y';
        pDBCArea->i_req_id = CallReqNo;

        DBCHCL(&Result, pContext, pDBCArea);
        if (Result != EM_OK) return;
        Flavor = pDBCArea->fet_parcel_flavor;
    }

```

```

/* Put this parcel into the trace table */
pDBCArea->func          = DBFIRQ;
pDBCArea->change_opts    = 'Y';
pDBCArea->using_data_ptr = (char *) &Flavor;
pDBCArea->using_data_len = sizeof(Flavor);

if (pDBCArea->fet_parcel_flavor == PclSUCCESS)
{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'Success')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclDATAINFO)
{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'DataInfo')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclXDIX)
{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'DataInfoX')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclRECORD)
{
    /* Point to the integer in the record parcel, */
    /* skip past the indicator byte                */
    pChar = pDBCArea->fet_data_ptr;
    pChar++;
    pInt32 = (Int32 *) pChar;
    sprintf(ReqStr,
        "USING (a INT) INSERT tracetable(:,a,'Record %d ')",
        *pInt32);
    pDBCArea->req_ptr = ReqStr;
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclENDSTATEMENT)
{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'EndStatement')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclENDREQUEST)

```

```

{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'EndRequest')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclRESULTSET)
{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'ResultSet')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else if (pDBCArea->fet_parcel_flavor == PclFAILURE)
{
    pChar = pDBCArea->fet_data_ptr;
    pChar += 4;
    pInt32 = (Int32 *) pChar;
    pChar += 4;
    for(a=0;a<strlen(pChar);a++)
        if(pChar[a]==0x27) pChar[a]=' ';
    sprintf(ReqStr,
        "USING (a INT) INSERT tracetable(:,a,'Failure %d %s')",
        (*pInt32)&0x0000ffff, pChar);
    pDBCArea->req_ptr = ReqStr;
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
else
{
    pDBCArea->req_ptr =
        "USING (a INT) INSERT tracetable(:,a,'')";
    pDBCArea->req_len = (Int32) strlen(pDBCArea->req_ptr);
}
DBCHCL(&Result, pContext, pDBCArea);
if (Result != EM_OK) return;

/* Fetch the success etc from the insert into the trace table */
pDBCArea->func = DBFFET;
pDBCArea->change_opts = 'Y';
pDBCArea->i_req_id = pDBCArea->o_req_id;
while (Result == EM_OK)
    DBCHCL(&Result, pContext, pDBCArea);

/* End the insert request */
pDBCArea->func = DBFERQ;
DBCHCL(&Result, pContext, pDBCArea);

```

```

        /* Go and fetch the next parcel from the CALL */
    }
}

/*****
    Entry point for the external stored procedure.
*****/
void sqlxspex(
    char      sqlstate[6],
    SQL_TEXT  extname[129],
    SQL_TEXT  specific_name[129],
    SQL_TEXT  error_message[257])
{
    Initialize();

    Logon(NULL);

    DoTheWork();

    Logoff();
}

```

## Executing SQL in Java External Stored Procedures

This example shows how to execute SQL in a Java external stored procedure.

The external stored procedure inserts data into a table that is defined like this:

```

CREATE TABLE webtab
  (id INTEGER
   ,webdata CLOB(1000K));

```

## JAR File Registration

The following statements register the JAR file for the external stored procedure with the JXSP database, creating an identifier for the JAR file called Web\_JAR:

```

DATABASE JXSP;
CALL SQLJ.INSTALL_JAR('C:\java_xsp\Web.jar', 'Web_JAR', 0);

```

## SQL Definition for the External Stored Procedure

```
REPLACE PROCEDURE JXSP.GetWebData(IN c1  INTEGER,
                                   IN aaa VARCHAR(200))

LANGUAGE JAVA
MODIFIES SQL DATA
PARAMETER STYLE JAVA
EXTERNAL NAME 'Web_JAR:Web.getWebData';
```

## Sample Query

The following query calls the external stored procedure:

```
CALL GetWebData(101, 'http://company.com');
```

To see the data that the external stored procedure stores on the parcels that it receives after calling the stored procedure, use the following statement:

```
SELECT * FROM webtab ORDER BY 1;
```

## Java Method Implementation

Here are the contents of a file called Web.java that you can compile to generate the file Web.class that you can place into a JAR file called Web.jar.

```
import com.teradata.fnc.*;
import java.net.*;
import java.io.*;
import java.sql.*;

public class Web {
    public static void getWebData(int ID, String url) throws SQLException
    {
        final int ClobSize = 1024*1000;
        final char[] buffer = new char[ ClobSize ];

        try {
            /* Establish default connection. */
            Connection con =
                DriverManager.getConnection( "jdbc:default:connection" );

            /* Prepare the command */
```

```

String sql = "INSERT INTO webtab ( ? , ? )";
PreparedStatement stmt = con.prepareStatement( sql );
stmt.setInt( 1, ID );

/* Get the data */
URL urlcon = new URL( url );
InputStreamReader data =
    new InputStreamReader(urlcon.openStream());
int length = data.read( buffer );
stmt.setCharacterStream(2, data, length );

/* Insert the data. */
stmt.execute();
stmt.close();
}
catch (Exception e) {
    throw new SQLException(e.getMessage(), "38U01");
}
}
}

```

## Example: Java External Stored Procedure With a Period Parameter

The following Java external stored procedure takes a Period type parameter and uses JDBC to insert it into a table. The value returned from the SELECT statement is returned as the OUT parameter.

```

//*****
//
// File:      JXSP_Period.java
// Header:    none
// Purpose:   Demonstrate a JXSP pdt_proc1 with SQL access.
//           The JXSP will:
//           - Take a Period(DATE) IN Parameter
//           - Connect as user guest/please
//           - Insert the Period(DATE) passed in into a table
//           - Select the Period(DATE) column from the table
//           - The value returned by SELECT is set as the
//             OUT parameter value.
//
// JDBC API: java.sql.Connection, java.sql.Statement,
//           java.sql.Statement.executeUpdate
//

```

```
//
//*****

import java.sql.*;
import java.io.*;

public class JXSP_Period
{
    // Name of the user able to create, drop, and manipulate tables
    public static String sUser = "guest";
    public static String sPassword = "please";

    /* REPLACE PROCEDURE PDT_PROC1(IN P1 PERIOD(DATE), OUT P2 PERIOD(DATE))
       LANGUAGE JAVA
       NO SQL
       PARAMETER STYLE JAVA
       EXTERNAL NAME 'JXSP_NEWTYPES:JXSP_Period.pdt_proc1';

       CREATE TABLE PerTypes(i int, P period(date));

    */

    public static void pdt_proc1(java.sql.Struct p1, java.sql.Struct p2[]) throws
    SQLException
    {
        // Creation of URL to be passed to the JDBC driver
        String url = "jdbc:teradata://whomooz/TMODE=ANSI,CHARSET=UTF8";
        try
        {
            // Loading the Teradata JDBC driver
            Class.forName("com.teradata.jdbc.TeraDriver");

            // Creating a connection object
            Connection con=DriverManager.getConnection(url,sUser, sPassword);
            try
            {
                String insertStmt = " INSERT INTO PerTypes ( 1 ,?,)";
                String selectStmt = " SELECT * FROM PerTypes ORDER BY 1";

                // Create a statement object from an active connection.
                Statement stmt = con.createStatement();

                try
                {

```

```

// Creating a prepared statement object
System.out.println("\n Preparing: " + insertStmt);
PreparedStatement pStmt =
    con.prepareStatement(insertStmt);

try
{
    // Call the following method to insert rows into the
    // sample table
insertRows(con, pStmt, p1); // Insert the IN parameter value

    // The following code will perform a SELECT query
    // on the table.
    stmt = con.createStatement();

    // Submit a query, creating a result set object
    ResultSet rs = stmt.executeQuery(selectStmt);

    // iterate through all returned rows and display them
    // (should be only 1 row)
    while (rs.next())
    {
        // retrieve the PERIOD ( DATE ) Struct
        Struct perDt = (Struct) rs.getObject(2);

        //Set the output value
        p2[0] = perDt;

    } // End while
}
finally
{
    // close the PreparedStatement
    pStmt.close();
    System.out.println(
        "\n PreparedStatement object closed.\n");
}

}
finally
{
    // close the statement
    stmt.close();
}
}

```



```

    }
    finally
    {
        //Close the connection
        System.out.println(" Closing Connection");
        con.close();

    }
}
catch (SQLException ex)
{
    // An SQLException was generated.  Catch it and display
    // the error information.
    // Note that there could be multiple error objects chained
    // together.
    System.out.println();
    System.out.println("*** SQLException caught ***");

    while (ex != null)
    {
        System.out.println(" Error code: " + ex.getErrorCode());
        System.out.println(" SQL State: " + ex.getSQLState());
        System.out.println(" Message: " + ex.getMessage());
        ex.printStackTrace();
        System.out.println();
        ex = ex.getNextException();
    }

    throw new IllegalStateException (" Sample failed." ) ;
}
}

public static void insertRows(Connection con, PreparedStatement pStmt,
    Struct period) throws SQLException
{
    // The following code will perform an INSERT query
    // on the sample table.
    System.out.println("\n Inserting Row...");
    pStmt.setInt (1, 1) ;
    pStmt.setObject (2, period) ;
    pStmt.executeUpdate () ;
} // End insertRows

```

```

public static void dropObject(Connection con, String dropQuery)
{
    try
    {
        // The following code will be used to perform a DROP query
        Statement stmt = con.createStatement();
        System.out.println(" Executing command "+dropQuery+"...");
        stmt.executeUpdate(dropQuery);
        System.out.println(" Command executed successfully.");
    }
    catch (SQLException ex)
    {
        // If the table did not exist, no drop is required.
        // Ignore the raised "no table present" exception by
        // printing out the error message and swallowing the
        // exception.
        System.out.println(" Ignoring exception: " + ex);
    }
} // End dropObject

} // End of class JXSP_Period

```

# UDM Code Examples

This section provides C code examples of UDMs for distinct and structured UDTs.

## Instance Method for a Distinct Type

This example shows the C code for a *toInches* instance method associated with a *meter* distinct UDT. The *toInches* instance method takes the *meter* value and returns the equivalent number of inches.

## SQL Definition

The following SQL statements show the DDL for the *meter* distinct UDT and the *toInches* method.

```
CREATE TYPE meter AS FLOAT
  FINAL
  INSTANCE METHOD toInches()
    RETURNS FLOAT
    SPECIFIC meter_toInches
    NO SQL
    PARAMETER STYLE TD_GENERAL
    DETERMINISTIC
    LANGUAGE C;

CREATE METHOD toInches()
  RETURNS FLOAT
  FOR meter
  EXTERNAL NAME 'CS!toinches!udm_src\to_inches.c!F!meter_toInches';
```

## Sample Query

The following statements show how to use the method in a query.

```
CREATE TABLE distance (pkey INTEGER, meters meter);
SELECT meters.toInches() from distance;
```

## C Function Definition

```
/****** C source file name: to_inches.c *****/

#define SQL_TEXT Latin_Text
```

```

#include <sqltypes_td.h>
#include <string.h>

void meter_toInches( UDT_HANDLE *meterUdt,
                    FLOAT *result,
                    char sqlstate[6])
{
    FLOAT value;
    int length;

    /* Get meter's value. */
    FNC_GetDistinctValue(*meterUdt, &value, sizeof_FLOAT, &length);

    /* Convert meters to inches and set the result value. */
    *result = value * 3.28 * 12;
}

```

## Related Information

For information on the FNC\_GetDistinctValue library function and the sizeof\_FLOAT macro, see [C Library Functions](#).

## Cast Method for a Distinct Type

This example shows the C code for a method that casts a *meter* distinct UDT to a *foot* distinct UDT. The method takes the *meter* value, calculates the equivalent number of feet, and returns the value as a *foot* distinct UDT.

## SQL Definition

The following SQL statements show the DDL for the *foot* distinct UDT, the *meter* distinct UDT, and the *toFoot* method that casts from a *meter* distinct UDT to a *foot* distinct UDT.

```

CREATE TYPE foot AS FLOAT FINAL;

CREATE TYPE meter AS FLOAT
    FINAL
    INSTANCE METHOD toFoot()
        RETURNS foot
        SPECIFIC meter_toFoot
        NO SQL
        PARAMETER STYLE TD_GENERAL
        DETERMINISTIC

```

```

LANGUAGE C;

CREATE METHOD toFoot()
  RETURNS foot
  FOR meter
  EXTERNAL NAME 'CS!tofoot!udm_src\to_foot.c!F!meter_toFoot';

CREATE CAST (meter AS foot)
  WITH SPECIFIC METHOD meter_toFoot AS ASSIGNMENT;

```

## Sample Query

Consider the following *metric\_distance* table that has a *meter* column and *distance* table that has a *foot* column:

```

CREATE TABLE metric_distance(pkey INTEGER, meters meter);
CREATE TABLE distance(pkey INTEGER, feet foot)

```

The following statement shows when Vantage uses the *meter\_toFoot* cast method in a query that requires implicit conversion from *meter* to *foot*:

```

INSERT INTO distance
  SELECT * FROM metric_distance;

```

## C Function Definition

```

/***** C source file name: to_foot.c *****/

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void meter_toFoot( UDT_HANDLE *meterUdt,
                  UDT_HANDLE *resultFootUdt,
                  char         sqlstate[6])
{
  FLOAT value;
  int length;

  /* Get meter's value. */
  FNC_GetDistinctValue(*meterUdt, &value, sizeof_FLOAT, &length);

  /* Convert meters to feet and set the result value. */

```

```

value *= 3.28;
FNC_SetDistinctValue(*resultFootUdt, &value, sizeof_FLOAT);
}

```

## Related Information

For information on the FNC\_GetDistinctValue and FNC\_SetDistinctValue library functions and the sizeof\_FLOAT macro, see [C Library Functions](#).

## Methods for Structured Type

This example shows the C code for UDMs that implement the following functionality for a structured UDT:

- constructor method for creating an initializing an instance of the UDT
- transform functionality that allows exporting the UDT from the server
- ordering functionality that allows two UDTs to be compared
- cast functionality that allows data conversion between the UDT and other data types

## SQL Definition of the UDT

Here is the SQL definition of a `color_t` distinct UDT and a `circle_t` structured UDT. Vantage automatically generates transform, cast, and ordering functionality for the `color_t` distinct UDT.

```

CREATE TYPE color_t
  AS VARCHAR(30)
  FINAL;

CREATE TYPE circle_t
  AS (x INTEGER, y INTEGER, radius INTEGER, color color_t)
  NOT FINAL
  CONSTRUCTOR METHOD circle_t( x INTEGER,
                               y INTEGER,
                               radius INTEGER,
                               color color_t )

  RETURNS circle_t
  SPECIFIC circle_t_constructor
  SELF AS RESULT
  NO SQL
  PARAMETER STYLE TD_GENERAL
  DETERMINISTIC
  LANGUAGE C,
  INSTANCE METHOD circle_t_FromSql()
  RETURNS VARCHAR(80)

```

```

    SPECIFIC circle_t_FromSql
    NO SQL
    PARAMETER STYLE TD_GENERAL
    DETERMINISTIC
    LANGUAGE C,
INSTANCE METHOD circle_t_Ordering()
    RETURNS FLOAT
    SPECIFIC circle_t_Ordering
    NO SQL
    PARAMETER STYLE SQL
    DETERMINISTIC
    LANGUAGE C,
INSTANCE METHOD VarcharCast()
    RETURNS VARCHAR(80)
    SPECIFIC circle_t_Cast
    NO SQL
    PARAMETER STYLE TD_GENERAL
    DETERMINISTIC
    LANGUAGE C;

```

## SQL Definition for the Constructor Method

The following CREATE METHOD statement installs the constructor method in the database:

```

CREATE CONSTRUCTOR METHOD circle_t(x INTEGER,
                                   y INTEGER,
                                   radius INTEGER,
                                   color color_t )

    RETURNS circle_t
    FOR circle_t
    EXTERNAL NAME 'CS!c_t!udmsrc/c_circle_t.c!F!circle_t_constructor';

```

## SQL Definitions for the Transform Method

The following CREATE METHOD statement installs the transform method in the database:

```

CREATE METHOD circle_t_FromSql()
    RETURNS VARCHAR(80)
    FOR circle_t
    EXTERNAL NAME 'CS!c_fromsql!udmsrc/c_fromsql.c!F!circle_t_FromSql';

```

You must add the *circle\_t\_ToSql* UDF before executing the following CREATE TRANSFORM statement. For the definition and C code of the *circle\_t\_ToSql* UDF, see [C Scalar Functions for UDT Functionality](#).

The following statement registers a UDF called *circle\_t\_ToSql* and the *circle\_t\_FromSql* UDM as transform routines for the *circle\_t* UDT:

```
CREATE TRANSFORM FOR circle_t
  circle_t_IO (TO SQL WITH SPECIFIC FUNCTION circle_t_ToSql,
    FROM SQL WITH SPECIFIC METHOD circle_t_FromSql);
```

## SQL Definitions for the Ordering Method

The following CREATE METHOD statement installs the ordering method in the database:

```
CREATE METHOD circle_t_Ordering()
  RETURNS FLOAT
  FOR circle_t
  EXTERNAL NAME 'CS!c_order!udmsrc/c_order.c!F!circle_t_Ordering';
```

The following statement registers the *circle\_t\_Ordering* UDM as an ordering routine for the *circle\_t* UDT:

```
CREATE ORDERING FOR circle_t
  ORDER FULL BY MAP WITH SPECIFIC METHOD circle_t_Ordering;
```

## SQL Definitions for the Cast Method

The following CREATE METHOD statement installs the cast method in the database:

```
CREATE METHOD VarcharCast()
  RETURNS VARCHAR(80)
  FOR circle_t
  EXTERNAL NAME 'CS!c_cast!udmsrc/c_cast.c!F!circle_t_Cast';
```

The following statement registers the *VarcharCast* UDM as a cast routine for the *circle\_t* UDT:

```
CREATE CAST (circle_t AS VARCHAR(80))
  WITH SPECIFIC METHOD circle_t_Cast
  AS ASSIGNMENT;
```

## Sample Queries

Consider the following table that defines a *circle\_t* column:

```
CREATE TABLE circleTbl( c_id INTEGER, circles circle_t);
```



To create and initialize an instance of the `circle_t` UDT using the `circle_t` constructor method defined previously in this example, you can use the `NEW` expression:

```
INSERT circleTbl( 1001, NEW circle_t(512, 512, 36, 'RED') );
```

You can also create and initialize an instance of the `circle_t` UDT like this:

```
INSERT circleTbl( 1002, NEW circle_t().circle_t(256, 128, 78, 'BLUE') );
```

The preceding statement invokes the `circle_t` constructor UDF that Vantage automatically generates for a structured UDT to create an instance of the `circle_t` UDT that has all of the attributes set to `NULL`. Then, the statement invokes the `circle_t` constructor method on the newly created instance of `circle_t` to initialize the attributes.

Here is an example where Vantage invokes the cast method to convert the UDT to a `VARCHAR(80)` type:

```
SELECT c_id
FROM circleTbl
WHERE CAST (circles AS VARCHAR(80)) = '256:128:78:BLUE';
```

Here is an example where Vantage invokes the transform method to export the data in the `circles` column from the server:

```
SELECT * FROM circleTbl;
```

Here is an example where Vantage invokes the following methods:

- Transform method to export the data in the `circles` column from the server
- Ordering method to compare the UDTs in the `circles` column
- Constructor method to create an instance of a `circle_t` UDT to use in the comparison

```
SELECT * FROM circleTbl WHERE circles < NEW circle_t(0,0,20,'RED');
```

## C Function Definition for the Constructor Method

The following code example implements the constructor method:

```
/* File: c_circle_t.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_constructor( UDT_HANDLE *circleUdt,
```

```

        INTEGER    *x_center,
        INTEGER    *y_center,
        INTEGER    *radius,
        UDT_HANDLE *colorUdt,
        UDT_HANDLE *resultCircleUdt,
        char        sqlstate[6])
{
    INTEGER x, y;
    int nullIndicator;
    VARCHAR_LATIN color[31];
    int length;
    int len;

    /* Set the x and y coordinates in the result circle UDT. */
    nullIndicator = 0;
    FNC_SetStructuredAttribute(*resultCircleUdt, "x", x_center,
nullIndicator, sizeof_INTEGER);
    nullIndicator = 0;
    FNC_SetStructuredAttribute(*resultCircleUdt, "y", y_center,
nullIndicator, sizeof_INTEGER);

    /* Set the radius. */
    nullIndicator = 0;
    FNC_SetStructuredAttribute(*resultCircleUdt, "radius", radius,
nullIndicator, sizeof_INTEGER);

    /* Get the color UDT value and set it in the result circle UDT. */
    FNC_GetDistinctValue(*colorUdt, color,
sizeof_VARCHAR_LATIN_WITH_NULL(30), &length);
    nullIndicator = 0;
    len = strlen(color);
    FNC_SetStructuredAttribute(*resultCircleUdt, "color", color,
nullIndicator, sizeof_VARCHAR_LATIN(len));
}

```

## C Function Definition for the Ordering Method

The following example implements ordering functionality that allows Vantage to order two *circle\_t* UDTs by their area. The function gets the radius and computes the area, returning the result as a FLOAT data type. If the circle is null or if the x attribute, y attribute, or radius attribute of the circle is null, the function returns a null result.

```

/* File: c_order.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_Ordering( UDT_HANDLE *circleUdt,
                       FLOAT      *result,
                       int         *indicator_circle,
                       int         *indicator_result,
                       char        sqlstate[6],
                       SQL_TEXT    extname[129],
                       SQL_TEXT    specific_name[129],
                       SQL_TEXT    error_message[257])
{
    INTEGER x, y, r;
    int nullIndicator;
    int length;

    /* If circle is null, return null. */
    if (*indicator_circle == -1) {
        *indicator_result = -1;
        return;
    }

    /* Verify that the x attribute is not null. */
    FNC_GetStructuredAttribute(*circleUdt, "x", &x, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == -1) {
        *indicator_result = -1;
        return;
    }

    /* Verify that the y attribute is not null. */
    FNC_GetStructuredAttribute(*circleUdt, "y", &y, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == -1) {
        *indicator_result = -1;
        return;
    }

    /* Get the radius attribute. */
    FNC_GetStructuredAttribute(*circleUdt, "radius", &r, sizeof_INTEGER,

```

```

&nullIndicator, &length);
    if (nullIndicator == -1) {
        *indicator_result = -1;
        return;
    }

    *result = 3.14 * r * r;

}

```

## C Function Definition for the Transform Method

The following example implements transform functionality that allows Vantage to export a *circle\_t* type as a VARCHAR(80). The format of the VARCHAR string is:

x:y:r:color

Null attributes are indicated by an absence of a value. For example, the format for a null radius is:

x:y::color

```

/* File: c_fromsql.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_FromSql( UDT_HANDLE    *circleUdt,
                      VARCHAR_LATIN *result,
                      char           sqlstate[6])
{
    INTEGER x, y, r;
    char y_str[11], r_str[11];
    VARCHAR_LATIN color[81];
    int nullIndicator;
    int length;

    /* Get each of the attributes and write them to the result string. */
    FNC_GetStructuredAttribute(*circleUdt, "x", &x, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == 0)
        sprintf((char *)result, "%d:", x);
    else
        strcpy((char *)result, ":");

```

```

    FNC_GetStructuredAttribute(*circleUdt, "y", &y, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == 0) {
        sprintf(y_str, "%d", y);
        strcat((char *)result, y_str);
    }
    strcat((char *)result, ":");

    FNC_GetStructuredAttribute(*circleUdt, "radius", &r, sizeof_INTEGER,
&nullIndicator, &length);
    if (nullIndicator == 0) {
        sprintf(r_str, "%d", r);
        strcat((char *)result, r_str);
    }
    strcat((char *)result, ":");

    FNC_GetStructuredAttribute(*circleUdt, "color", color,
sizeof_VARCHAR_LATIN_WITH_NULL(80), &nullIndicator, &length);
    if (nullIndicator == 0)
        strcat((char *)result, (char *)color);
}

```

## C Function Definition for the Cast Method

The following example implements cast functionality for converting a *circle\_t* UDT to a VARCHAR(80) predefined data type. The format of the VARCHAR string is:

x:y:r:color

Null attributes are indicated by an absence of a value. For example, the format for a null radius is:

x:y::color

```

/* File: c_cast.c */

#define SQL_TEXT Latin_Text
#include <sqltypes_td.h>
#include <string.h>

void circle_t_Cast( UDT_HANDLE      *circleUdt,
                   VARCHAR_LATIN *result,
                   char             sqlstate[6])
{
    INTEGER x, y, r;
    char y_str[11], r_str[11];

```

```

VARCHAR_LATIN color[81];
int nullIndicator;
int length;

/* Get each of the attributes and write them to the result string. */
FNC_GetStructuredAttribute(*circleUdt, "x", &x, sizeof_INTEGER,
&nullIndicator, &length);
if (nullIndicator == 0)
    sprintf((char *)result, "%d:", x);
else
    strcpy((char *)result, ":");

FNC_GetStructuredAttribute(*circleUdt, "y", &y, sizeof_INTEGER,
&nullIndicator, &length);
if (nullIndicator == 0) {
    sprintf(y_str, "%d", y);
    strcat((char *)result, y_str);
}
strcat((char *)result, ":");

FNC_GetStructuredAttribute(*circleUdt, "radius", &r, sizeof_INTEGER,
&nullIndicator, &length);
if (nullIndicator == 0) {
    sprintf(r_str, "%d", r);
    strcat((char *)result, r_str);
}
strcat((char *)result, ":");

FNC_GetStructuredAttribute(*circleUdt, "color", color,
sizeof_VARCHAR_LATIN_WITH_NULL(80), &nullIndicator, &length);
if (nullIndicator == 0)
    strcat((char *)result, (char *)color);
}

```

## Dropping the Objects Created in this Example

You can use the following SQL statements to drop the objects that were created for this example.

```

DROP CAST (circle_t AS VARCHAR(80));
DROP ORDERING FOR circle_t;
DROP TRANSFORM circle_t_io FOR circle_t;
DROP FUNCTION circle_t_tosql;

```

```
ALTER TYPE circle_t DROP SPECIFIC METHOD circle_t_constructor;  
ALTER TYPE circle_t DROP SPECIFIC METHOD circle_t_cast;  
ALTER TYPE circle_t DROP SPECIFIC METHOD circle_t_ordering;  
ALTER TYPE circle_t DROP SPECIFIC METHOD circle_t_fromsql;  
  
DROP TYPE circle_t;  
DROP TYPE color_t;
```

# C/C++ Command-line Debugging for UDFs

This section uses the name *Teradata C/C++ UDF Debugger* to refer to a version of GDB (the GNU Source-Level Debugger) that contains extensions implemented by Teradata. The Teradata C/C++ UDF Debugger supports standard GDB commands and adds commands for debugging protected-mode external routines written in C or C++:

- User-defined functions (UDFs): scalar, aggregate, table functions, and table operators
- External stored procedures
- User-defined methods (UDMs)

This document uses the term “UDFs” to refer collectively to UDFs, external stored procedures, and UDMs.

Online information about the GNU debugger, including the Teradata C/C++ UDF Debugger extensions is available from the Linux command line by typing `info tdgdb`.

Teradata also offers Eclipse (Studio) plug-in debuggers for C/C++ and Java. For more information on these debuggers, see [Teradata Debugger for C/C++ UDF](#) and [Teradata Debugger for Java UDF](#).

## Required Privileges

Privileges needed for creating and debugging UDFS:

- CREATE FUNCTION
- EXECUTE FUNCTION

Privileges needed for creating and debugging external stored procedures:

- CREATE EXTERNAL PROCEDURE
- EXECUTE PROCEDURE

Privileges needed for debugging UDFs or external stored procedures by a user other than the one who created the routines:

- EXECUTE FUNCTION or EXECUTE PROCEDURE
- DROP FUNCTION or DROP PROCEDURE

Privileges needed for creating and debugging UDMs:

- ALL on SYSUDTLIB

For example, the following statement grants all privileges on SYSUDTLIB to the user named debugger:

```
grant all on sysudtlib to debugger with grant option;
```

## Benefits of Teradata C/C++ UDF Debugger

Debugging UDFs poses a particularly challenging problem in the Teradata environment. Due to the massively parallel nature of Teradata, UDFs can run in multiple processes on multiple nodes.



Using a regular debugger (like the standard GDB) requires users to attach to each UDF server process manually, in separate debugging sessions. The Teradata C/C++ UDF Debugger attaches automatically to every UDF server associated with an SQL request and controls those UDFs from a single debugging session. These features provide modern debugging capabilities for UDFs running within the database.

## Teradata C/C++ UDF Debugger Capabilities

Database users with sufficient permissions can debug a UDF, an external stored procedure, or a UDM within a SQL session. On one database, multiple simultaneous SQL sessions may be started for debugging, either by the same or different users.

Users can join the Teradata C/C++ UDF Debugger only to their own SQL sessions. Nothing from other sessions is visible to them, and the debugger encrypts its message traffic to deter eavesdropping. After the debugger is joined to a session, it will see every instance of the UDF that is put into execution and is able to install breakpoints and monitor the UDF execution. One debugger can join multiple named UDFs if they execute from different SQL sessions but each session can debug only one named UDF at a time.

The debugger supports only external routines written in C or C++.

Debugging facilities available for UDFs include nearly everything GDB can do to debug normal programs, including:

- set and display breakpoints
- display variables
- watch variables or memory locations
- resume execution
- display the function call stack
- make breakpoints conditional
- display source code

There are also commands specific to controlling UDFs, such as:

- list UDFs available for debugging
- join UDFs to the debug session
- select among multiple simultaneously executing UDFs

## Recommended Test Environment

The debugger is intended for use only on development or test systems. Running it in a production environment raises a risk of debugging activities impacting critical workloads in a number of ways. Potential impacts include (but are not limited to) skewed performance measurement due to long-running UDFs, database locks held by UDFs being debugged, and extra memory consumption for debugger processing.

## Restrictions

The debugger supports external routines running in secure or protected mode. It does not support unprotected mode. UDFs meant to run in unprotected mode should be debugged only by running them in secure or protected mode. After they are working properly then alter them to run in unprotected mode.

## Simple Debugging Example

The topics in this section describe how to create a simple scalar UDF that calculates the sum of two integers, execute the UDF in protected mode, and debug it. The same high-level process can be used to debug any external routine that the debugger supports.

### Setting Up the System

Some setup is needed before actually debugging the UDF. First, you must enable UDF debugging to permit database users with sufficient permissions to debug a UDF. This option is disabled by default to deter misuse on production systems. The UDF Debugging option on the debug screen in the ctl utility enables it.

1. Start ctl from a command shell on the target system and go to the debug screen:

```
# ctl
> screen debug

(0) Start DBS:           On           (1) Break Stop:          Off
(2) Start With Logons: All           (3) Start With Debug:    Off
(4) Save Dumps:          Off          (5) Snapshot Crash:      Off
(6) Maximum Dumps:       -1           (7) Start PrgTraces:     Off
(8) Restart Dump Type: System         (9) UDF Debugging:       Off
```

2. If UDF Debugging is not already on, enable it and quit ctl:

```
> 9=on
> quit
CTL: Write Control GDO Changes? y
CTL: Control GDO successfully written.
Warning: A change has been made to one or more fields
        that has a deferred effect.
        TPA reset must be performed so that those changes can
        take effect.
```

3. As the message from ctl says, you must restart Teradata before the new setting takes effect. You can do that with a tpareset command at the shell prompt:

```
# tpareset -y Enable UDF Debugging
```

## Creating a Database User

Debugging requires that special privileges be given to the debugger user. This example creates a debugger user with permissions to create and execute UDFs. Any DBS user can debug UDFs, but every user that does so must have permission.

1. Login to bteq as dbc user. This example assumes you do this from a command shell on the target system, but bteq can run from any machine that can access the Teradata system.

From a shell command prompt, logon to bteq:

```
# bteq .logon localhost/dbc,dbc_password
```

2. Once bteq issues its command prompt, create a user named, "debugger":

```
create user debugger
as permanent=50e6,
spool=100e6,
temporary=50e6,
password=debugger;
```

3. Submit the following commands to grant this user the required privileges:

```
grant create function on debugger to debugger with grant option;
grant execute function on debugger to debugger with grant option;
```

4. Log out and quit.

## Creating the C UDF source file

After the debugger user is set up, you can use this user account to create a simple scalar UDF and install the function in the database. Here is a sample of C source code that you can use to create a source file named plusudf.c. The function calculates the sum of two integers.

1. For this example, create the plusudf.c file in the /tmp directory.

```
/*
 * plusudf.c
 *
 * Sample UDF for debugging
 */

#define SQL_TEXT Latin_Text
#include "sqltypes_td.h"

void plusudf(INTEGER *a, INTEGER *b, INTEGER *result, char sqlstate[5])
{
```

```
*result = *a + *b;
}
```

## Log Into bteq

1. Log into bteq as the debugger user:

```
# bteq .logon localhost/debugger,debugger
```

The UDF will be installed in the database to which you are logged on. The function is stored in a library that contains code for all the UDFs created by this user account. In this example the DBS is running on the local machine.

## Create the New UDF

1. Create the new UDF, named “plusudf,” from your C file using CREATE FUNCTION with the “d” option in the EXTERNAL NAME clause:

```
create function plusudf(
a integer,
b integer
) returns integer
language c
no sql
parameter style td_general
external name 'd!cs!plusudf!/tmp/plusudf.c';
```

You must use the “d” option so that the UDF is compiled with debug symbols. Every time you create or replace a UDF the library is rebuilt. For more information on CREATE FUNCTION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

## Verify the UDF

1. Run the UDF to ensure that it works properly:

```
select plusudf(5,10);
```

The results should show:

```
*** Query completed. One row found. Onlne column returned.
*** Total elapsed time was 2 seconds.

plusudf(5,10)
15
```

## Run the UDF to Be Debugged

After the UDF is created and verified, it can be run for debugging.

1. Log into a normal SQL session:

```
# bteq .logon localhost/debugger,debugger
```

2. Request that any subsequent SQL statements in the SQL session that invoke the UDF (named “plusudf” in this example) will run under the Teradata C/C++ UDF Debugger:

```
set session debug function plusudf on;
```

3. Issue an SQL statement that causes plusudf to run.

```
select plusudf(5,10);
```

The UDF halts the execution of the query and waits for the debugger to join the session. The query will not complete until the debugger allows the function to continue. If your query puts into execution multiple instances of the UDF, the query will not complete until the debugger has let all the UDFs finish running.

## Join the UDF to a Teradata C/C++ UDF Debugger Session

The debugger claims the UDF and controls it.

1. Start the debugger in a separate window on the target system. You can use either `tdgdb` or `/usr/pde/bin/gdb` from a shell prompt:

```
# tdbgdb
```

2. To start the debugging session, use the `attach` command with the `udf` option and the debugger account name and password:

```
(gdb) attach udf localhost/debugger,debugger
Attaching to UDF server via localhost/debugger,debugger
```

### Note:

The UDF server makes debugging UDFs running in the database possible for GDB (with the Teradata C/C++ UDF Debugger extensions).

3. To list the UDFs waiting to be debugged that can be joined by this debugger account, use the `info udf` command. This command produces something like:

```
(gdb) info udf
Sessno  Name      Type  Language  Count  Joined
-----
1001    plusudf   UDF   C          1      No
```

Column	Description
Sessno	SQL session number.
Name	Name of the UDF.
Type	Type is always UDF.
Language	Programming language in which function source code is written.
Count	Instance number of UDF being debugged.
Joined	Whether SQL session in which UDF is running has been joined by debugger.

4. Use the session number supplied by the info udf display to select the UDF to associate with the debugging session. Type:

```
join session number
```

For session 1001, the command output produces something like:

```
(gdb) join 1001
Reading symbols for task udfsectsk... (libudf.so, 0x7ffff7bb7000)
(libudf1.so, 0x7ffff79b4000) (libstdc++.so.6, 0x7ffff76a9000)
(libjil.so, 0x7ffff748b000) (libnetpde.so, 0x7ffff723f000)
(libpde.so, 0x7ffff6f80000) (libemf.so, 0x7ffff6d72000)
(libpdesym.so, 0x7ffff6b5b000) (libpthread.so.0, 0x7ffff693e000)
(libelf.so.1, 0x7ffff672a000) (libnsl.so.1, 0x7ffff6512000)
(libm.so.6, 0x7ffff62bc000) (libc.so.6, 0x7ffff5f5a000)
(libdl.so.2, 0x7ffff5d56000) (ld-linux-x86-64.so.2, 0x7ffff7dde000)
(libgcc_s.so.1, 0x7ffff5b3f000) (libacl.so.1, 0x7ffff5937000)
(libcrypto.so.0.9.8, 0x7ffff5598000)
(libthread_db.so.1, 0x7ffff5390000) (libattr.so.1, 0x7ffff518b000)

(libz.so.1, 0x7ffff4f75000) (libudf_1026_17.so, 0x7ffff355b000)
Node   Pid   Tid   Type
16383  7066   7066   C
```

Note that Node shows the vproc ID on which the UDF is running.

## Debug the UDF

Now you can debug using standard GDB commands to set breakpoints, display variables, and step or continue execution.

1. Use the dir command to tell the Teradata C/C++ UDF Debugger where the source file is located:

```
(gdb) dir /tmp
Source directories searched: /tmp:$cdire:$cwd
```

**Note:**

If the source file is not available, users with sufficient permissions can use SHOW FUNCTION in BTEQ to show the C source code for a UDF. The output can be saved to a file and used for debugging. For more information, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

2. Set a breakpoint in the file at line 12:

```
(gdb) br plusudf.c:12
Breakpoint 1 at 0x2aaaac25f714: file plusudf.c, line 12.
```

**Note:**

After the debugger has seen the library that contains code for the UDF, you can set breakpoints in that library regardless of whether the UDF is currently running. Any additional breakpoints will be included in new instances of the UDF as they start.

3. Type C to let the query run to the breakpoint:

```
(gdb) c
Continuing.
Breakpoint 1, plusudf (a=0x2aaaac21e940, b=0x2aaaac21e950,
    result=0x2aaaac21e964, sqlstate=0x2aaaac21e96c "00000") at plusudf.c:12
12          *result = *a + *b;
```

4. Print the current values of the variables:

```
(gdb) print *a
$1 = 5
(gdb) print *b
$2 = 10
```

5. Type c to continue.

In this example the query runs to completion in your bteq window.

```
(gdb) c
Continuing.
```

When a query is complete the session remains joined until you quit the debugger. If in BTEQ you were to reissue the same request in your SQL session a new UDF instance would display in the debugger window when the UDF hits the breakpoint.

- Press `Ctrl+c` to terminate the debugger command in progress and return to the GDB command prompt; then type `quit` to allow your query to run to completion and then exit the debugger.

```
<Ctrl-c>
Debug Server stopped without any debug processes!
(gdb) quit
Ending debugging session.
```

If your debugging uncovers the need for changes to your UDF, the best practice is to quit the debugger after you recompile a UDF and restart the debugger to reestablish whatever breakpoints are needed in the new debugging session, much the same as you would do to debug a stand-alone program.

You can update the source code and recompile it with a `REPLACE FUNCTION` statement in the same `bteq` session that you used to debug it. However, replacing the function causes the library to be rebuilt so any breakpoints you set on the initial UDF are not preserved.

## Parallel Debugging Example

This example introduces the parallel debugging facilities of the Teradata C/C++ UDF Debugger. When a SQL request executes a UDF, multiple instances of the UDF may be running on one or more AMPs. Using the Teradata C/C++ UDF Debugger, you can switch between the instances running on those AMPs. This example will refer collectively to those instances as *UDFs*.

The [Simple Debugging Example](#) uses a query that runs a single UDF to illustrate the basic debugging process. You can follow that example and reproduce exactly the steps in the process. The details in this example cannot be reproduced exactly. Results are dependent on your specific database system configuration. What you see may differ from what is shown here, but this example will illustrate some of the available parallel debugging capabilities.

### Create a Table

To produce similar results to this example, create a 2-column table, named “testdata,” loaded with these values:

a	b
1	2
2	1
3	4
4	3
5	6
6	5
7	8
8	7
9	10



You can create this table in bteq with this statement:

```
create table testdata ( a integer, b integer ) primary index ( a );
```

Populate this table with the rows shown above, either by entering them with INSERT statements in bteq or by loading them from a file using FastLoad.

For information on using BTEQ, see *Basic Teradata® Query Reference*, B035-2414. For information on using FastLoad, see *Teradata® FastLoad Reference*, B035-2411.

## Debug Multiple Instances of a UDF

Under the debugger, execute a request to run plusudf (the UDF you created in [Simple Debugging Example](#)) for each row in the testdata table, for example:

```
set session debug function plusudf on;
select a, b, plusudf(a, b) from testdata;
```

Follow the steps described in [Simple Debugging Example](#). After joining this query to the debugger you will see something like:

```
(gdb) join 1003
Reading symbols for task udfsectsk... (libudf.so, 0x7ffff7bb7000)
(libudf1.so, 0x7ffff79b4000) (libstdc++.so.6, 0x7ffff76a9000)
(libjil.so, 0x7ffff748b000) (libnetpde.so, 0x7ffff723f000)
(libpde.so, 0x7ffff6f80000) (libemf.so, 0x7ffff6d72000)
(libpdesym.so, 0x7ffff6b5b000) (libpthread.so.0, 0x7ffff693e000)
(libelf.so.1, 0x7ffff672a000) (libnsl.so.1, 0x7ffff6512000)
(libm.so.6, 0x7ffff62bc000) (libc.so.6, 0x7ffff5f5a000)
(libdl.so.2, 0x7ffff5d56000) (ld-linux-x86-64.so.2, 0x7ffff7dde000)
(libgcc_s.so.1, 0x7ffff5b3f000) (libacl.so.1, 0x7ffff5937000)
(libcrypto.so.0.9.8, 0x7ffff5598000)
(libthread_db.so.1, 0x7ffff5390000) (libattr.so.1, 0x7ffff518b000)
(libz.so.1, 0x7ffff4f75000) (libudf_1026_17.so, 0x7ffff355b000) done.
Node  Pid  Tid  Type
  3  5758  5758  C
  2  5748  5748  C
  1  5743  5743  C
  0  5753  5753  C
```

Instead of just one line in the table listing the UDFs joined, this now shows one line for each AMP—Node 3, 2, 1, 0. (Note that the debugger uses the label “Node” to display the vproc number. An AMP is a specific type of vproc.) If you set a breakpoint at line 12 of the UDF and continue, you will see something like this:

```
(gdb) b plusudf
Breakpoint 1 at 0x7ffff356d12c: file plusudf.c, line 12.
(gdb) c
Continuing.
0//5753: Secondary breakpoint #1 at plusudf() (plusudf.c line 12)
1//5743: Secondary breakpoint #1 at plusudf() (plusudf.c line 12)
3//5758: Secondary breakpoint #1 at plusudf() (plusudf.c line 12)
[Switching to 2//5748]

Breakpoint 1, plusudf (a=0x7ffff7fa1940, b=0x7ffff7fa1950, result=0x7ffff7fa1964,
sqlstate=0x7ffff7fa196c "00000")
    at plusudf.c:12
12          *result = *a + *b;
```

The example above shows that UDFs reach the breakpoint in all AMPs at the same time. Debugging must be performed serially on the UDFs. The breakpoint in vproc 2 was reached first, so the debugger switches the current context to that UDF (indicated by [Switching to 2//5748]). All the other UDFs that reached the breakpoint (on vproc 0, 1, and 3) are labeled as Secondary breakpoint and remain stopped at the breakpoint while you work on the thread in the current context.

When you display variables at this point, only the UDF in the current context (vproc 2, thread 57481) is affected:

```
(gdb) p *a
$1 = 7
(gdb) p *b
$2 = 8
(gdb) p *result
$3 = 0
```

When you step execution the function continues from the line it stopped at:

```
(gdb) n
13      }
(gdb) p *result
$4 = 15
```

You can see that the line executed and that printing `*result` returns the value for that UDF.

You can switch to one of the other UDFs waiting at a breakpoint with the context command (abbreviated as an @ sign) followed by the thread selector:

```
(gdb) @ 3//5758
[3//5758]
```

```
(gdb) p *a
$5 = 6
```

Note that after you select a new UDF and display an argument, it shows the value for the newly selected UDF. You can go through all UDFs currently stopped and step each one individually. You can also set context to all active UDFs and display a variable for all of them with one command:

```
(gdb) @ all
[1//5743]
(gdb) p *a
[1//5743]
$6 = 9
[0//5753]
$7 = 5
[2//5748]
$8 = 7
[3//5758]
$9 = 6
```

You can provide one or more specific thread selectors to set context to one or more specific tasks. When multiple tasks are in context, the value for each task is preceded by a line showing the value's thread selector. Although you can show values for multiple UDFs, you can only step execution in one UDF at a time. The thread selector that is displayed after the `@ all` command shows which UDF that will be.

Continuing execution resumes all currently active UDFs, which in this example run to completion and have a result similar to the following:

```
(gdb) c
Continuing.
[Switching to 0//5753]
Breakpoint 1, plusudf (a=0x7ffff7fa1940, b=0x7ffff7fa1950, result=0x7ffff7fa1964,
sqlstate=0x7ffff7fa196c "00000")
    at plusudf.c:12
12          *result = *a + *b;
```

Notice that in this example, the UDF reaches the breakpoint on vproc 0, which is the same vproc on which a UDF completed before but for a different table row. Secondary breakpoints could occur here if multiple UDFs hit the breakpoint at the same time, but only one UDF hit it in this instance.

At this point, you can display variables and step through the new UDF as before:

```
(gdb) p *a
$10 = 3
(gdb) p *b
$11 = 4
```

```
(gdb) n
13      }
(gdb) p *result
$12 = 7
```

Whenever the debugger stops, it only reports UDFs that have hit a breakpoint, but it suspends execution of all UDFs that it has joined. To see the others, you can put all threads in context (@ all) and issue the info context long command to display all threads in context. At this point in the debug session, that might show something like:

```
(gdb) @ all
[2//5748]
(gdb) i context long
all -> 2//5748 0//5753
```

UDF 2//5748 was not reported when the debugger reached the breakpoint. It was joined to the debugger but had not yet hit its breakpoint when all the UDFs were stopped. Printing UDF variables for 2//5748 will not work because execution stopped inside the line that defines the variables:

```
(gdb) p *a
[2//5748]
No symbol "a" in current context.
[0//5753]
$15 = 3
```

Continuing again at this point allows 2//5748 to run to its breakpoint:

```
(gdb) c
Continuing.
Breakpoint 1, plusudf (a=0x7ffff7fa1940, b=0x7ffff7fa1950, result=0x7ffff7fa1964,
sqlstate=0x7ffff7fa196c "00000")
    at plusudf.c:12
12      *result = *a + *b;
(gdb) info context
2//5748
```

No message about switching context appears here because the current context set by the previous @ command already contains the thread that stopped. Showing the context after the stop confirms which thread stopped, in this case 2//5748.

Continuing again will allow this thread to run, and the debugger will stop when one or more UDFs reach the breakpoint. You can continue until the UDF has executed for every row in the table. If you do not want to step through every UDF, you can quit the debugger and the query will then run to completion without stopping.

Alternately, you can delete or disable the breakpoint and continue to let the query finish without leaving the debugger. At that point, rerunning the query will stop the query again when the first UDF runs.

Parallel debugging offers many more capabilities than this simple example illustrates. If you only want to step through problem code that only a few UDFs execute, setting a breakpoint at that point will only stop UDFs that hit it; all other UDFs will silently execute without stopping. You can also set conditions on breakpoints to stop only when data values of interest are passed to a UDF. Commands can be attached to breakpoints that automatically display variables whenever breakpoints hit; the last command can continue execution to generate a trace of UDFs that a query executes without stopping at all.

## Differences Between Standard GDB and Teradata C/C++ UDF Debugger Commands

The Teradata C/C++ UDF Debugger extends GDB; therefore all GDB commands that are available for normal debugging are also available for debugging UDFs. Keep in mind that some operations are not safe (or possible) in a massively parallel environment, such as reverse execution and catching certain system calls (such as fork and exec).

When you use the Teradata C/C++ UDF Debugger, all threads attached to the debugger stop when any thread encounters a condition that stops it. The `continue` command runs all threads (unless they are held); the `step` and `next` commands only run the current thread. In contrast, some versions of GDB stop only the threads that encounter a breakpoint (or other condition that will stop them) and all other threads continue to run.

### kill

The `kill` command aborts the current database request when issued in a UDF debugging session, not the attached process as in a normal GDB session. If more than one database session is joined to a debugging session, the `kill` command ends debugging of any active SQL requests from all joined sessions, but only the request for the current UDF is aborted. Database sessions can then restart queries to continue debugging their UDFs.

### info thread

The `info thread` command lists all threads currently known to GDB. When debugging UDFs, it shows each thread selector in the Target Id field and the thread's current instruction address. This can be useful to identify other UDFs that are running when one stops at a breakpoint.

## Teradata Commands for UDF Debugging

The following Teradata-specific commands are available only to GDB with the Teradata C/C++ UDF Debugger extensions:

- [attach udf](#)
- [context](#)
- [delete watchpoint](#)

- [hold](#)
- [info context](#)
- [info held](#)
- [info map](#)
- [info scope](#)
- [info udf](#)
- [info watch](#)
- [join](#)
- [scope](#)
- [unhold](#)
- [watch](#)

Some of these commands use thread selectors as parameters or in their output. A thread selector consists of three numeric values separated by a slash (/) character. Thread selectors take the form *node/process/thread*, where *node* is the number of the vproc that invoked the UDF, and *process* and *thread* are the identifiers assigned by the operating system to the process and thread that executes the UDF. Together the node and thread uniquely identify every task on the system, so process is normally omitted. For example, a thread selector for a PE thread might be 16383//7137; 0//22638 could be an AMP thread.

Thread selectors in UDF debugging sessions typically correspond to UDFs being debugged. However, udfsectsk processes that execute the UDFs have threads in addition to the one that executes the UDF. The debugger normally hides those threads but they become visible if they hit a breakpoint, watch point, or a signal that the debugger catches. When that happens, those threads are handled the same as UDF threads and are assigned thread selectors.

## attach udf

This command is used to create a Teradata C/C++ UDF Debugger session on the requested system.

### Syntax

```
attach udf systemname/username,password
```

### Syntax Elements

#### **systemname**

The name of the Teradata system to log onto as a debugger.

#### **username**

The name for the user account to use to log on.

The username must be same as the username used to run the UDF that is to be debugged.

***password***

The password for the account.

**Usage Notes**

`attach udf` may be issued before or after you start the UDF to be debugged in `bteq`; however, you must create the Teradata C/C++ UDF Debugger session before you issue other UDF-specific debugging commands. This command attaches the debugger to the database, not to a specific database session.

**context**

This command specifies the threads on which subsequent commands to the debugger will operate.

**Syntax**

```
context all
context thread-selector [...]
```

**Syntax Elements*****thread-selector***

Specifies a string that selects one or more UDF threads being debugged.

**Usage Notes**

You can specify one or more specific threads, or all threads to include all currently active threads. When the context contains multiple threads, commands operate on all of them serially. The output for each thread contains a line that identifies the thread followed by a line of command-related information. After a breakpoint or other exception stops a thread, the context defaults to that thread making it the current context.

**Note:**

You can use `@` as a synonym for the context command. If context is used, it must be fully spelled out.

**delete watchpoint**

This command deletes an individual watchpoint specified by its watch slot number. Any watchpoint currently defined for that watch slot is removed.

**Syntax**

```
delete watch n
```

## Syntax Elements

*n*

Watchpoint number to delete (0, 1, 2, or 3). `watch/d n` is an equivalent command.

## hold

You can use the `hold` command to control which UDFs run when `continue` commands are entered. Threads that are held will not execute; all others will execute.

### Syntax

```
hold all
hold thread-selector [...]
```

## Syntax Elements

*thread-selector*

Specifies a string that selects one or more UDF threads being debugged.

## Usage Notes

You can use `hold` and `unhold` together. For example, you could use `hold all` and then `unhold` just the thread you want to execute.

`hold` affects only the `continue` command; the `step` and `next` commands implicitly hold all threads except the current one.

## info context

This command displays the current context on which the debugger will next operate.

### Syntax

```
info context [ long ]
```

## Syntax Elements

*long*

Displays all threads in the current context. Useful when context is set to `all`. Appends a list of the threads in context.



## info held

This command displays the current hold status. If no threads are held then no status is returned.

### Syntax

```
info held [ long ]
```

### Syntax Elements

long

Appends a list of threads that are being held from execution.

### Examples: Using the info held Command

This example shows using info held after hold all and the difference between info held and info held long:

```
(gdb) hold all
(gdb) i held
all
(gdb) i held long
all -> 1//6441 0//6457
```

This example shows using info held after a hold of specific threads:

```
(gdb) hold 1//6441 0//6457
(gdb) i held
( 1//6441 0//6457 )
(gdb) i held long
( 1//6441 0//6457 ) -> 1//6441 0//6457
```

When changes to a hold are made using the unhold command, info held long displays the original hold command parameters followed by a minus sign and a list of released threads. It also displays the resulting threads that are still held.

```
(gdb) hold 1//6441 0//6457
(gdb) i held long
( 1//6441 0//6457 ) -> 1//6441 0//6457
(gdb) unhold 0//6457
(gdb) i held long
( 1//6441 0//6457 ) - ( 0//6457 ) -> 1//6441
```

## info map

This command displays a memory usage map. For each segment in the address space of the process being debugged, it shows its base address, size, access permissions and optionally, the name of the file if this memory is a mapped file.

### Syntax

```
info map
```

## info scope

This command displays the current scope on which the debugger will next operate.

### Syntax

```
info scope [ long ]
```

### Syntax Elements

long

Appends a list of all threads in the scope.

## info udf

This command shows the names of the UDFs that are attached to the database and are available for debugging.

### Syntax

```
info udf [ wait ]
```

### Syntax Elements

wait

Requests that this command not complete until there is at least one UDF available to debug. The wait can be interrupted by pressing **Ctrl+C**.

### Usage Notes

The output of this command appears in a table that provides a row for each UDF that is available to be debugged by the requesting database. It contains the following UDF information:

- session number
- function name

- function type
- source programming language
- number of instances
- whether the UDF is currently joined to a debugger session

The table usually has only one row, but can have more rows if the user is running multiple debugging sessions.

## info watch

This command displays information for the current watchpoints for the current thread. A thread can have up to four watched addresses. The watched locations directly reference the Intel debug x8664 registers. The locations are thread specific so they are not sensitive to the scope setting.

### Syntax

```
info watch
```

### Example: Sample Results From the info watch Command

Here is an example of the info watch results:

```
Debug Registers: (Single Step Trap)
Hit RW SZ LG
0:  T  W  4  L 00002aaaac4a0964
1:      R  4  L 00002aaaac4a0950
2:      X  1  0000000000000000
3:      X  1  0000000000000000
```

Column	Description
Hit	T if watchpoint has hit, blank otherwise.
RW	Mode specified for watchpoint—R, W, or X. X indicates unused slot.
SZ	Size of watchpoint location in bytes—1, 2, 4, or 8 bytes. 1 indicates unused slot.
LG	L for local, G for global. Always L.

## join

This command joins a UDF to a debugging session.

## Syntax

```
join sessno[/udfname] [ wait ]
```

## Syntax Elements

### *sessno*

The session number of the UDF to be debugged. This is typically taken from the output of a prior `info udf` command, but UDFs can be joined without a prior `info udf` command if the user already knows the number for the session to debug.

### *udfname*

The name of a UDF waiting to be debugged. This is optional.

### *wait*

The optional wait parameter requests that this command not complete until there is at least one UDF available to debug.

Joining a session number without specifying *udfname* joins any UDF debugged by that session. Joining a specific *udfname* only joins that UDF.

## Usage Notes

At least one UDF that matches the join criteria must be waiting to be debugged when the command is entered. However, once joined, all current and future instances of the named UDF (or all UDFs) from the specified session are caught by this debugging session.

Multiple `join` commands can be issued to debug multiple UDFs, or even multiple database sessions. Once something is joined, though, it can only be unjoined by terminating the debugger session. Anything joined to one debugger session cannot be joined to another one until you exit the first session.

## scope

This command selects the UDF instances that will have breakpoints installed when the UDFs are continued.

## Syntax

```
scope all
scope { thread-selector [...] | all }
```

## Syntax Elements

### *thread-selector*

Specifies a string that selects one or more UDF threads being debugged.

Default: all (applies breakpoints to all instances)

## Usage Notes

Note that breakpoint scope applies only for the duration of the UDF invocation for which it is set. When a UDF in scope exits, it is removed from the scope. If all of the UDFs in scope exit, then the scope is reset to all.

New UDFs with the same thread selectors do not inherit the scope that was set previously. To limit breakpoints to specific UDFs, it may work better to hold off running multiple UDFs than to set scope to the one of interest.

## unhold

You can use the `unhold` command to control which UDFs run when `continue` commands are entered. Threads that are held will not execute; all others will execute.

## Syntax

```
unhold all
unhold thread-selector [...]
```

## Syntax Elements

### *thread-selector*

Specifies a string that selects one or more UDF threads being debugged.

## Usage Notes

You can use `hold` and `unhold` together. For example, you could use `hold all` and then `unhold` just the thread you want to execute.

This affects only the `continue` command; the `step` and `next` commands implicitly hold all threads except the current one.

## watch

This command replaces the standard GDB `watch` command, which is not suitable for multi-threaded, multi-process, distributed execution. A thread can have up to four watched addresses. Defining a watchpoint uses the next available of the four watch slots. This command directly allocates a hardware debug register for the current thread such that when the condition occurs on the specified address, the thread will stop.

This applies on a per-thread basis. Watchpoints are implicitly removed when the UDF for which they are set exits. They must be reestablished for each UDF, even if it runs on a thread that also ran a previous UDF.

## Syntax

```
watch[/mode] location
```

## Syntax Elements

### *mode*

Specifies the kind of memory access to watch for:

- d deletes a watchpoint
- r memory read access
- w memory write access
- x instruction execute access

The number of bytes to watch defaults to the size of the location specified, but that can be overridden by preceding the mode letter with 1, 2, 4, or 8. Those are the only sizes that can be watched.

A mode of 'd' deletes a previously specified watchpoint. Instead of a location to watch, this must be followed by a watchpoint number (0, 1, 2 or 3).

Although hardware execution watchpoints are supported, they are rarely used because normal GDB breakpoints are generally more effective.

### *location*

An expression that evaluates to the address of the location that is to be monitored.

## Usage Notes

You can make watchpoints apply to every UDF with a breakpoint set on entry to the function. This sets the desired watchpoint and continues:

```
(gdb) break funcname
Breakpoint 1 at ...
(gdb) command 1
> silent
> watch location/w
> continue
>end
(gdb)
```

# Procedure to Enable R Functionality with ExecR

## Note:

Due to the complexity of installation, length of time and need for a Database restart, Teradata strongly recommends enlisting Teradata Customer Services to install the R Components and Packages. Please contact your Teradata representative to order this service.

After an R interpreter is installed on a Vantage system, R can be used to run R scripts through either the SCRIPT table operator or the ExecR table operator.

To use the ExecR table operator, in addition to the R interpreter, the following are required:

- The udfGPL library
- You must run the optional DIP script DipRTblOp to create the ExecR table operator, which is an alternative Vantage facility to run R scripts.

When the R interpreter is installed through a Change Control request with Customer Services, the process also includes the installation of the udfGPL library and activation of ExecR, and thus satisfies the above requirements.

If the R interpreter is installed by your system administrator on your system, follow the steps below to complete the tasks. A root user access and the database DBC user password on the Advanced SQL Engine nodes are required.

## Determining the Running Advanced SQL Engine Database Version

Run the following command in a Bash shell of the target system to determine the database version of the running Advanced SQL Engine:

```
pdepath -i
```

You can also submit the following query to the target system database to determine the database version of the running Advanced SQL Engine:

```
SELECT * from dbc.dbcinfo;
```

For example, the following results from the query show that the database version is 16.20.33.01.

InfoKey	InfoData
VERSION	16.20.33.01
RELEASE	16.20.33.01

## Checking if the udfGPL Package is Installed

Log on to a Advanced SQL Engine node and submit the following command to check whether the package `teradata-udfgpl` exists on the system.

```
rpm -qa | grep udfgpl
```

If present, then continue to [Creating the ExecR System Table Operator](#).

Otherwise, [download](#) and [install](#) the package.

## Downloading the udfGPL Package

1. Click **Login** to sign on to Teradata Support portal at <https://support.teradata.com>.
2. Under **Tools**, click **Software Downloads**.
3. In the left pane, click **Database and Applications**.
4. In the main pane, under **Database and Applications**, click **Teradata Database**.
5. In the main pane, select the following:

Field	Value
Teradata Releases	<b>16.20</b>
Target OSs	The OS version running on the TPA node.
Current	<b>All</b>

Click **Submit**.

6. Determine the Certification Date for your running Advanced SQL Engine database version:
  - a. Scroll down and find `ptdbms` in the Package Name column.
  - b. Locate the row for your running database version, and click **Cert Lookup**.
  - c. In the **Certification Lookup** window, note the **Certification Date**.
7. Find the Teradata packages based on the Certification Date:
  - a. From the left navigation pane, under **Certification Lists**, click **Previous Lists**.
  - b. For the **Certification List** fields, select the following:

Field	Value
Node Type	<b>TPA</b>
Target OSs	The OS version running on the TPA node.
Teradata Release	The running Advanced SQL Engine release.
BusType	<b>SM3G</b>



Field	Value
Certification Date	The Certification Date you retrieved from step 6.
Would you like to compare lists?	<b>No</b>

Click **Submit**.

8. Scroll down to find the `teradata-udfopl` in the Package Name column, and select it to download.
9. Scroll down to the end of the table and select the following download options:

Field	Value
Download Type	<b>Public FTP (<a href="ftp.teradata.com">ftp.teradata.com</a>)</b>
Download Readme Files?	<b>Yes</b>

10. Enter your **User Name**, **E-Mail**, **Site ID**, and **Change Control Number**.
11. In the **FTP/SFTP/PFTP Directory** field, enter the name of the FTP directory where the files will be placed.
12. Click **Submit**.

---

**Note:**

You will receive an email that includes the URL and security information you need to retrieve the files from the FTP site.

---

13. Place each downloaded package into the `/tmp` directory on the primary (PDN) node.
- 

**Note:**

You must obtain the `teradata-udfopl` package that matches your version of the TDBMS from step A above.

---

## Installing the udfGPL Library

---

**Note:**

You must install the R interpreter before installing the `teradata-udfopl` package.

---

TDR is the R package interface between R and the Advanced SQL Engine. Both TDR and the udfGPL library are included in the `teradata-udfopl` package. Each version of the TDBMS will have a compatible version of this package.

The `ctl` utility does not list this optional package. You must use PUT to install it separately. See [Installing Packages Using PUT or the rpm Tool](#).

TDR is compiled and installed during a Advanced SQL Engine database version switch or installed during rpm installation if the running TDBMS matches the udfGPL version.

R table operators use the udfGPL secure server. The **GPLUDFServerMemSize** field in the cufconfig GDO (Globally Distributed Object) is used to limit the memory of the udfGPL server. This limit is set to 32 MB by default, but you can set the limit from 0 to 3.5 GB. To use the maximum available memory, set GPLUDFServerMemSize to 0.

For more information, see [Configuring the udfGPL Server Memory Limit](#).

## Installing Packages Using PUT or the rpm Tool

You can install software packages on all database nodes using the Parallel Upgrade Tool (PUT) or manually using the rpm tool.

### Installing Packages Using PUT

#### Prerequisites:

- A minimum PUT version must be installed on your database system for compliance with Java requirements. PUT will let you know if a version upgrade is needed.
- Java must be enabled on your web browser.
- The packages to install must be placed in advance in the `/var/opt/teradata/customermodepkgs` directory of your target database PDN node.

Use PUT to install the rpms on all nodes of your database system.

1. From your browser, connect to the target database PDN node by typing the PDN node IP address or name in the address line.

For example, if your system is named `vantageSystem` with the PDN node called `vantageSystem1`, then enter the following on your browser address line:

```
vantageSystem1:8080/put
```

2. Log in with the target system root user credentials.
3. When the **PUT** screen appears, select **Install/Upgrade Software**.
4. Proceed through the next couple of screens by clicking the **Next** button. Namely, acknowledge the screen that informs you PUT is running in Customer Mode; then, on the following screen ignore a message that might pop up and protest that PUT is unable to find any packages (click **OK** on the pop up window, if it appears).
5. On the package selection screen, click sequentially on all packages you may want to install.
6. For each package you select, you will be taken to a screen to select the nodes to install this package on. Check whether all your system nodes are under the **Selected** area. If not, then in the left pane, **Select All** available nodes on your system and move them to the **Selected** area using the **>>** button. Select **Selected**.
7. Installation takes place, and you are notified when the task is finished.
8. Select **Next** and you can exit the PUT tool.

**Note:**

If for any reason any missing dependency issues should arise in the process, simply obtain the required packages in the fashion illustrated in the [Downloading the udfGPL Package](#), place them in the /var/opt/teradata/customermodepkgs directory of your target Advanced SQL Engine database PDN node together with the other packages to install, and repeat this installation process.

**Installing Packages Using the rpm Tool**

Install the teradata-udfgpl packages that matches your version of the TDBMS on all nodes of the database by first copying the package in the /tmp directory on every node of your system, using the command:

```
pc1 -send /tmp/teradata-udfgpl-16.20.xx.xx-1.0.x86_64.rpm /tmp
```

Then, run the following command:

```
psh rpm -U /tmp/teradata-udfgpl-16.20.xx.xx-1.0.x86_64.rpm
```

**Creating the ExecR System Table Operator**

The ExecR system table operator is used to execute R scripts. Run the optional Database Initialization Program (DIP) script DipRTbOp to create the ExecR table operator. The DipRTbOp DIP script is not run as part of the DIPALL script. You must run it separately.

ExecR resides in the td\_sysgpl database.

For information about DIP, see *Teradata Vantage™ - Database Utilities*, B035-1102.

The following is an example of how to run the DipRTbOp script from a Advanced SQL Engine primary (PDN) node:

```
#cnstern 6
Input Supervisor Command:
> start dip
Started 'dip' in window 1

Input Supervisor Command:
> ^C -- <control C>
#cnstern 1

Attempting to connect to CNS...Completed
Type the password for user DBC or press the Enter key to quit:
> xxx
***Logon successfully completed.

Select one of the following DIP SQL scripts to execute:
(Press the Enter key to quit)
```

```

...
40. DIPRTBLOP - R Table Operator

> 40
Executing DIPRTBLOP
Please wait...

DIPRTBLOP is complete
Please review the results in /var/opt/teradata/tdtemp/dip40.txt on node 1-1

Would you like to execute another DIP script (Y/N)?
> N

Exiting DIP...

```

## Configuring the udfGPL Server Memory Limit

As mentioned in the [Installing the udfGPL Library](#), you can adjust the memory limit of the udfGPL server, within which ExecR operates.

Run the `cufconfig` GDO (Globally Distributed Object) utility and look for the **GPLUDFServerMemSize** field, if you wish to modify the default 32 MB memory limit of the udfGPL server. Recommended values for the **GPLUDFServerMemSize** field are in the range of 1 - 3 GB (1073741824 to 3221225472 bytes). Lack of memory might result in errors during ExecR execution.

The following example assumes default settings on a system, and shows how to check and change the **GPLUDFServerMemSize** value to 1 GB.

1. Connect as root user to the target system, and use a Bash shell command line for the following.
2. Inspect the current `cufconfig` parameters and their values by executing the command:

```
/usr/tdbms/bin/cufconfig -o
```

3. Locate the **GPLUDFServerMemSize** field in the output listing. It should appear among the output as follows:

```

...
  GPLUDFServerMemSize: 33554432
...

```

This means the **GPLUDFServerMemSize** size is 33554432 bytes / 1024 (KB/byte) / 1024 (MB/KB) = 32 MB.

4. To change this value to 1 GB, create a text file in the `/tmp` directory of every node on the server, so that this file contains the line: `GPLUDFServerMemSize: 1073741824`, using the following command:

```
psh "printf 'GPLUDFServerMemSize: 1073741824' > /tmp/ccChange.txt"
```

---

**Note:**

Memory size must be specified in bytes.

---

5. Invoke the `cufconfig` utility to accept the new setting by executing the following command:

```
psh /usr/tdbms/bin/cufconfig -i -f /tmp/ccChange.txt
```

6. You can inspect the result of your action by typing again, using the following command:

```
/usr/tdbms/bin/cufconfig -o
```

---

**Note:**

Changes in the **GPLUDFServerMemSize** field take effect the following time an ExecR query is executed.

---

# Additional Information

## Teradata Links

Link	Description
<a href="https://docs.teradata.com/">https://docs.teradata.com/</a>	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
<a href="https://support.teradata.com">https://support.teradata.com</a>	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none"><li>• Community support</li><li>• Software updates</li><li>• Knowledge articles</li></ul>
<a href="https://www.teradata.com/University/Overview">https://www.teradata.com/University/Overview</a>	Teradata education network
<a href="https://support.teradata.com/community">https://support.teradata.com/community</a>	Link to Teradata community