

# **Teradata Vantage™ - SQL Operators and User- Defined Functions**

---

Release 17.10

July 2021

# Copyright and Trademarks

Copyright © 2019 - 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 Operators and User-Defined Functions</b>	<b>5</b>
Changes and Additions	5
<b>Chapter 2: Compression/Decompression Functions</b>	<b>6</b>
CAMSET	6
CAMSET_L	9
DECAMSET	12
DECAMSET_L	13
JSON_COMPRESS	14
JSON_DECOMPRESS	16
LZCOMP	17
LZCOMP_L	20
LZDECOMP	22
LZDECOMP_L	23
TD_LZ_COMPRESS	25
TD_LZ_DECOMPRESS	27
TS_COMPRESS	28
TS_DECOMPRESS	30
TransUnicodeToUTF8	32
TransUTF8ToUnicode	34
<b>Chapter 3: Export Width Procedures</b>	<b>36</b>
About Export Width	36
ReplaceExportDefinition	37
RemoveExportDefinition	40
<b>Chapter 4: File System Information Macros and Functions</b>	<b>42</b>
CreateFsysInfoTable/CreateFsysInfoTable_ANSI	42
PopulateFsysInfoTable/PopulateFsysInfoTable_ANSI	45
AlterFsysInfoTable_TD16/AlterFsysInfoTable_ANSI_TD16	84
Heatmap Table Function and Macro (tdheatmap and tdheatmap_m)	91
<b>Chapter 5: Map Functions, Macros, and Procedures</b>	<b>111</b>
Map Functions	111
Map Macros	116
Map Procedures	116
<b>Chapter 6: Table Operators</b>	<b>157</b>
CALCMATRIX	157

Cogroups .....	163
FeatureNames_TBF .....	165
R Table Operator .....	167
READ_NOS .....	169
SCRIPT .....	189
TD_DBQLParam .....	204
TD_DBQLFUL .....	207
TD_UNPIVOT .....	209
WRITE_NOS .....	213
<b>Chapter 7: User-Defined Functions .....</b>	<b>225</b>
Scalar UDF .....	225
Aggregate UDF .....	229
Window Aggregate UDF .....	232
User-Defined Functions .....	240
UDF Invocation .....	243
<b>Chapter 8: User-Defined Type Expressions/Methods .....</b>	<b>244</b>
UDT Expression .....	244
NEW .....	248
NEW JSON .....	251
NEW VARIANT_TYPE .....	251
NEW XML .....	255
Method Invocation .....	255
<b>Chapter 9: Script Installation Procedures .....</b>	<b>260</b>
Execution Rules .....	260
About Installing and Registering External Language Scripts .....	260
About Replacing External Language Scripts .....	262
About Redistributing External Language Scripts .....	264
Displaying the User-Installed File .....	265
About Removing External Language Scripts .....	265
SYSUIF.REMOVE_FILE .....	265
Example .....	266
<b>Appendix A: Notation Conventions .....</b>	<b>267</b>
<b>Appendix B: Additional Information .....</b>	<b>270</b>

# Introduction to SQL Operators and User-Defined Functions

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.

This document describes SQL functions, methods, macros, and procedures useful for database administrators. These functions give you the ability to do the following:

- Compress and uncompress data
- Manage the amount of data exported to client applications
- Find information on specific data blocks and see how often objects on specific AMPs are accessed
- Use maps to optimize the placement of tables across AMPs or redistribute table rows after a system expansion
- Use Teradata-created UDFs and UDTs
- Install, replace, redistribute, and remove script files
- Use table operators to do these things:
  - Manipulate tables
  - Run scripts to convert DBQL data to JSON
  - Run Linux and R programs inside the database

## Changes and Additions

Date	Description
July 2021	<ul style="list-style-type: none"> <li>• Added new examples to show the LOCAL line in the PopulateFsysInfoTable macro output.</li> <li>• Added AUTHORIZATION syntax element to READ_NOS.</li> <li>• Added READNOS_SCHEMA option to RETURNTYPE syntax element of READ_NOS.</li> <li>• Added Google Native Connector support to READ_NOS.</li> <li>• Fallback subtables are no longer compressed by default.</li> <li>• Updated Azure LOCATION syntax.</li> <li>• Updated WRITE_NOS ACCESS_ID and ACCESS_KEY information.</li> </ul>
June 2020	<ul style="list-style-type: none"> <li>• Updated the Installation of R Components and Packages under R Table Operator to support the R in-database distribution.</li> <li>• Updated Setting Memory Limits under SCRIPT to show the steps to configure the ScriptMemLimit field in GDO to limit available memory.</li> <li>• Added READ_NOS table operator, which allows access to external files in JSON or CSV format.</li> </ul>

# Compression/Decompression Functions

The following sections describe the functions that you can use with algorithmic compression (ALC) to compress and uncompress column data of character or byte type. Compression of data reduces space usage and the size of disk I/Os.

If the compression and decompression functions described here are not optimal for your data, you can write your own user-defined functions (UDFs) to compress and uncompress table columns.

## Note:

Using ALC together with block-level compression (BLC) may degrade performance, so this practice is not recommended.

## Prerequisites

The functions described here are embedded services system functions. For information on activating and invoking embedded services functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.

FOR more information on...	SEE...
ALC	<ul style="list-style-type: none"> <li>• COMPRESS and DECOMPRESS phrases and CREATE TABLE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i>, B035-1144.</li> </ul>
writing UDFs for ALC	<ul style="list-style-type: none"> <li>• Information about defining functions for algorithmic compression in <i>Teradata Vantage™ - SQL External Routine Programming</i>, B035-1147.</li> </ul>
compression methods supported by Vantage and a comparison of the various methods	<ul style="list-style-type: none"> <li>• <i>Teradata Vantage™ - Database Design</i>, B035-1094.</li> </ul>

## CAMSET

Compresses the specified Unicode character data into the following possible values using a proprietary Teradata algorithm:

- Partial byte values (for example, 4-bit digits or 5-bit alphabetic letters)
- One byte values (for example, other Latin characters)
- Two byte values (for example, other Unicode characters)

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
[TD_SYSFNLIB.] CAMSET ( Unicode_string )
```

### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *Unicode\_string*

A Unicode character string or string expression.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET UNICODE, where the maximum supported size (*n*) is 32000. You can also pass arguments with data types that can be converted to VARCHAR(32000) CHARACTER SET UNICODE using the implicit data type conversion rules that apply to UDFs. For example, CAMSET(CHAR) is allowed because it can be implicitly converted to CAMSET(VARCHAR).

### Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

The input to this function must be Unicode character data.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARBYTE(64000).

## Usage Notes

Uncompressed character data in Vantage requires 2 bytes per character when storing Unicode data. CAMSET takes Unicode character input, compresses it into partial byte, one byte, or two byte values, and returns the compressed result.

CAMSET provides best results for short or medium Unicode strings that:

- contain mainly digits and English alphabet letters.
- do not frequently switch between:
  - lowercase and uppercase letters.
  - digits and letters.
  - Latin and non-Latin characters.

Although you can call the function directly, CAMSET is normally used with algorithmic compression (ALC) to compress table columns. If CAMSET is used with ALC, nulls are also compressed if those columns are nullable.

## Restrictions

CAMSET currently can only compress Unicode characters from U+0000 to U+00FF.

## Uncompressing Data Compressed with CAMSET

To uncompress Unicode data that was compressed using CAMSET, use the DECAMSET function.

## Examples

### Example: Compressing Unicode Values

In this example, the Unicode values in the Description column are compressed using the CAMSET function with ALC. The DECAMSET function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET UNICODE,
   Description VARCHAR(1000) CHARACTER SET UNICODE
    COMPRESS USING TD_SYSFNLIB.CAMSET
    DECOMPRESS USING TD_SYSFNLIB.DECAMSET);
```

### Example: Querying for Compressed Values

Given the following table definition:

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Description VARCHAR(100) CHARACTER SET UNICODE);
```



The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.CAMSET(Pendants.Description);
```

## Related Information

- For more information about the COMPRESS and DECOMPRESS phrases, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about UDF implicit type conversion rules, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information about the DECAMSET function, see [DECAMSET](#).
- For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Teradata Vantage™ - Database Design*, B035-1094.

## CAMSET\_L

Compresses the specified Latin character data into the following possible values using a proprietary Teradata algorithm:

- Partial byte values (for example, 4-bit digits or 5-bit alphabetic letters)
- One byte values (for example, other Latin characters)

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
[TD_SYSFNLIB.] CAMSET_L ( Latin_string )
```

### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *Latin\_string*

A Latin character string or string expression.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET LATIN, where the maximum supported size (*n*) is 64000. You can also pass arguments with data types that can be converted to VARCHAR(64000) CHARACTER SET LATIN using the implicit data type conversion rules that apply to UDFs. For example, CAMSET\_L(CHAR) is allowed because it can be implicitly converted to CAMSET\_L(VARCHAR).

---

### Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

---

The input to this function must be Latin character data.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARBYTE(64000).

## Usage Notes

Uncompressed character data in Vantage requires one byte per character when storing Latin character data. CAMSET\_L takes Latin character input, compresses it into partial byte or one byte values, and returns the compressed result.

CAMSET\_L provides best results for short or medium Latin strings that:

- contain mainly digits and English alphabet letters.
- do not frequently switch between:
  - lowercase and uppercase letters.
  - digits and letters.

Although you can call the function directly, CAMSET\_L is normally used with algorithmic compression (ALC) to compress table columns. If CAMSET\_L is used with ALC, nulls are also compressed if those columns are nullable.

## Uncompressing Data Compressed with CAMSET\_L

To uncompress Latin character data that was compressed using CAMSET\_L, use the DECAMSET\_L function.

## Examples

### Example: Compressing Latin Values

In this example, the Latin values in the Description column are compressed using the CAMSET\_L function with ALC. The DECAMSET\_L function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPERCASE CHARACTER SET LATIN,
   Description VARCHAR(1000) CHARACTER SET LATIN
    COMPRESS USING TD_SYSFNLIB.CAMSET_L
    DECOMPRESS USING TD_SYSFNLIB.DECAMSET_L);
```

### Example: Querying for Compressed Latin Values

Given the following table definition:

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Description VARCHAR(100) CHARACTER SET LATIN);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.CAMSET_L(Pendants.Description);
```

## Related Information

- For more information about the COMPRESS/DECOMPRESS phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about ALC, see COMPRESS and DECOMPRESS phrases in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about UDF implicit type conversion rules, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information about the DECAMSET function, see [DECAMSET](#).
- To uncompress Latin character data that was compressed using CAMSET\_L, use the DECAMSET\_L function. See [DECAMSET\\_L](#).
- For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Teradata Vantage™ - Database Design*, B035-1094.

## DECAMSET

Uncompresses the Unicode data that was compressed using the CAMSET function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] DECAMSET ( compressed_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *compressed\_string*

Unicode character data that was compressed using the CAMSET function.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the CAMSET function.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARCHAR(32000) CHARACTER SET UNICODE.

## Usage Notes

DECAMSET takes Unicode data that was compressed using the CAMSET function, uncompresses it, and returns an uncompressed Unicode character string as the result.

Although you can call the function directly, DECAMSET is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with CAMSET.

## Example: Compressing Unicode Values

In this example, the Unicode values in the Description column are compressed using the CAMSET function with ALC. The DECAMSET function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET UNICODE,
   Description VARCHAR(1000) CHARACTER SET UNICODE
   COMPRESS USING TD_SYSFNLIB.CAMSET
   DECOMPRESS USING TD_SYSFNLIB.DECAMSET);
```

## Related Information

For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## DECAMSET\_L

Uncompresses the Latin data that was compressed using the CAMSET\_L function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] DECAMSET_L ( compressed_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *compressed\_string*

Latin character data that was compressed using the CAMSET\_L function.

## Argument Type and Rules

Expressions passed to this function must have a data type of `VARBYTE(n)`, where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the `CAMSET_L` function.

If you specify `NULL` as input, the function returns `NULL`.

## Result Type

The result data type is `VARCHAR(64000) CHARACTER SET LATIN`.

## Usage Notes

`DECAMSET_L` takes Latin data that was compressed using the `CAMSET_L` function, uncompresses it, and returns an uncompressed Latin character string as the result.

Although you can call the function directly, `DECAMSET_L` is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with `CAMSET_L`.

## Example

In this example, the Latin values in the Description column are compressed using the `CAMSET_L` function with ALC. The `DECAMSET_L` function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET LATIN,
   Description VARCHAR(1000) CHARACTER SET LATIN
    COMPRESS USING TD_SYSFNLIB.CAMSET_L
    DECOMPRESS USING TD_SYSFNLIB.DECAMSET_L);
```

## Related Information

- For more information about the `COMPRESS/DECOMPRESS` phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about ALC, see the `COMPRESS` and `DECOMPRESS` phrases in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## JSON\_COMPRESS

Compresses JSON data type values.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
[TD_SYSFNLIB.] JSON_COMPRESS ( JSON_expr )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *JSON\_expr*

An expression that evaluates to a JSON data type.

---

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

---

## Argument Type and Rules

This function only compresses JSON data types. It cannot be used to compress any other data type. Users cannot create their own UDFs for compressing JSON data.

## Result Type

The function accepts TD\_ANYTYPE as input and returns TD\_ANYTYPE so it can be used on any form of the JSON data type.

You can use the TD\_LZ\_COMPRESS function to compress JSON data; however, Teradata recommends that you use JSON\_COMPRESS instead because the JSON\_COMPRESS function is optimized for compressing JSON data.

## Example: Compressing JSON Data Types

Although you can call the function directly, JSON\_COMPRESS is often used with the COMPRESS USING phrase to compress table columns. In the following table, the JSON data in the json\_col column is compressed using the JSON\_COMPRESS function.

```
CREATE TABLE temp (
  id          INTEGER,
  json_col    JSON(1000)
```

```
CHARACTER SET LATIN
COMPRESS USING JSON_COMPRESS
DECOMPRESS USING JSON_DECOMPRESS);
```

## Related Information

- For more information about the COMPRESS USING phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about the COMPRESS/DECOMPRESS phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## JSON\_DECOMPRESS

Uncompresses the JSON data previously compressed using the JSON\_COMPRESS function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] JSON_DECOMPRESS ( compressed_JSON_data )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *compressed\_JSON\_data*

JSON data that was compressed using the JSON\_COMPRESS function.

## Argument Type and Rules

This function only uncompresses JSON data types. It cannot be used to uncompress any other data type. Users cannot create their own UDFs for compressing and uncompressing JSON data.



## Result Type

The return data type of the function is JSON.

## Example: Decompressing JSON Data Types

Although you can call the function directly, JSON\_DECOMPRESS is often used with the DECOMPRESS USING phrase to uncompress table columns. In the following table, the JSON data in the json\_col column is uncompressed using the JSON\_DECOMPRESS function.

```
CREATE TABLE temp (
  id          INTEGER,
  json_col    JSON(1000)
              CHARACTER SET LATIN
              COMPRESS USING JSON_COMPRESS
              DECOMPRESS USING JSON_DECOMPRESS);
```

## Related Information

- For more information about the DECOMPRESS USING phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about the COMPRESS/DECOMPRESS phrase, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## LZCOMP

Compresses the specified Unicode character data using the Lempel-Ziv algorithm.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] LZCOMP ( Unicode_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### Unicode\_string

A Unicode character string or string expression.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET UNICODE, where the maximum supported size (*n*) is 32000. You can also pass arguments with data types that can be converted to VARCHAR(32000) CHARACTER SET UNICODE using the implicit data type conversion rules that apply to UDFs. For example, LZCOMP(CHAR) is allowed because it can be implicitly converted to LZCOMP(VARCHAR).

---

### Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

---

For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

The input to this function must be Unicode character data.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARBYTE(64000).

## Usage Notes

Uncompressed character data in Vantage requires 2 bytes per character when storing Unicode data. LZCOMP takes Unicode character input, compresses it using the Lempel-Ziv algorithm, and returns the compressed result.

LZCOMP provides good compression results for long Unicode strings, but might not be as effective for short strings. It can also provide good results for medium strings that have many repeating characters.

Although you can call the function directly, LZCOMP is normally used with algorithmic compression (ALC) to compress table columns. If LZCOMP is used with ALC, nulls are also compressed if those columns are nullable.

## Uncompressing Data Compressed with LZCOMP

To uncompress Unicode data that was compressed using LZCOMP, use the LZDECOMP function.

### Examples

#### Example: Compressing Unicode Values

In this example, the Unicode values in the Description column are compressed using the LZCOMP function with ALC. The LZDECOMP function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET UNICODE,
   Description VARCHAR(1000) CHARACTER SET UNICODE
    COMPRESS USING TD_SYSFNLIB.LZCOMP
    DECOMPRESS USING TD_SYSFNLIB.LZDECOMP);
```

#### Example: Querying for the Compressed Unicode Values of the Description column

Given the following table definition:

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Description VARCHAR(100) CHARACTER SET UNICODE);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.LZCOMP(Pendants.Description);
```

### Related Information

- For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Teradata Vantage™ - Database Design*, B035-1094.
- See for <http://zlib.net> information about the compression algorithm used by LZCOMP.
- To uncompress Unicode data that was compressed using LZCOMP, use the LZDECOMP function. See [LZDECOMP](#).

## LZCOMP\_L

Compresses the specified Latin character data using the Lempel-Ziv algorithm.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] LZCOMP_L ( Latin_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *Latin\_string*

A Latin character string or string expression.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET LATIN, where the maximum supported size (*n*) is 64000. You can also pass arguments with data types that can be converted to VARCHAR(64000) CHARACTER SET LATIN using the implicit data type conversion rules that apply to UDFs. For example, LZCOMP\_L(CHAR) is allowed because it can be implicitly converted to LZCOMP\_L(VARCHAR).

#### Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.

The input to this function must be Latin character data.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARBYTE(64000).

## Usage Notes

Uncompressed character data in Vantage requires one byte per character when storing Latin character data. LZCOMP\_L takes Latin character input, compresses it using the Lempel-Ziv algorithm, and returns the compressed result.

LZCOMP\_L provides good compression results for long Latin character strings, but might not be as effective for short strings. It can also provide good results for medium strings that have many repeating characters.

Although you can call the function directly, LZCOMP\_L is normally used with algorithmic compression (ALC) to compress table columns. If LZCOMP\_L is used with ALC, nulls are also compressed if those columns are nullable.

## Uncompressing Data Compressed with LZCOMP\_L

To uncompress Latin data that was compressed using LZCOMP\_L, use the LZDECOMP\_L function.

## Examples

### Example: Compressing Latin Values

In this example, the Latin values in the Description column are compressed using the LZCOMP\_L function with ALC. The LZDECOMP\_L function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET LATIN,
   Description VARCHAR(1000) CHARACTER SET LATIN
    COMPRESS USING TD_SYSFNLIB.LZCOMP_L
    DECOMPRESS USING TD_SYSFNLIB.LZDECOMP_L);
```

### Example: Querying for the Compressed Latin Values of the Description column

Given the following table definition:

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Description VARCHAR(100) CHARACTER SET LATIN);
```

The following query returns the compressed values of the Description column.

```
SELECT TD_SYSFNLIB.LZCOMP_L(Pendants.Description);
```

## Related Information

- For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- To uncompress Latin data that was compressed using LZCOMP\_L, use the LZDECOMP\_L function. See [LZDECOMP\\_L](#).
- See <http://zlib.net> for information about the compression algorithm used by LZCOMP\_L.
- For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Teradata Vantage™ - Database Design*, B035-1094.
- For details, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## LZDECOMP

Uncompresses the Unicode data that was compressed using the LZCOMP function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] LZDECOMP ( compressed_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *compressed\_string*

Unicode character data that was compressed using the LZCOMP function.

## Argument Type and Rules

Expressions passed to this function must have a data type of `VARBYTE(n)`, where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the `LZCOMP` function.

If you specify `NULL` as input, the function returns `NULL`.

## Result Type

The result data type is `VARCHAR(32000) CHARACTER SET UNICODE`.

## Usage Notes

`LZDECOMP` takes Unicode data that was compressed using the `LZCOMP` function, uncompresses it, and returns an uncompressed Unicode character string as the result.

Although you can call the function directly, `LZDECOMP` is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with `LZCOMP`.

## Example: Compressing Unicode Values with LZCOMP

In this example, the Unicode values in the `Description` column are compressed using the `LZCOMP` function with ALC. The `LZDECOMP` function uncompresses the previously compressed values.

```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET UNICODE,
   Description VARCHAR(1000) CHARACTER SET UNICODE
    COMPRESS USING TD_SYSFNLIB.LZCOMP
    DECOMPRESS USING TD_SYSFNLIB.LZDECOMP);
```

## Related Information

- For more information about the `COMPRESS` and `DECOMPRESS` phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- See <http://zlib.net> for information about the decompression algorithm used by `LZDECOMP`.

## LZDECOMP\_L

Uncompresses the Latin data that was compressed using the `LZCOMP_L` function.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
[TD_SYSFNLIB.] LZDECOMP_L ( compressed_string )
```

### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *compressed\_string*

Latin character data that was compressed using the LZCOMP\_L function.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the LZCOMP\_L function.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARCHAR(64000) CHARACTER SET LATIN.

## Usage Notes

LZDECOMP\_L takes Latin data that was compressed using the LZCOMP\_L function, uncompresses it, and returns an uncompressed Latin character string as the result.

Although you can call the function directly, LZDECOMP\_L is normally used with algorithmic compression (ALC) to uncompress table columns previously compressed with LZCOMP\_L.

## Example: Compressing Latin Values with LZCOMP\_L

In this example, the Latin values in the Description column are compressed using the LZCOMP\_L function with ALC. The LZDECOMP\_L function uncompresses the previously compressed values.



```
CREATE MULTISET TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPER CASE CHARACTER SET LATIN,
   Description VARCHAR(1000) CHARACTER SET LATIN
    COMPRESS USING TD_SYSFNLIB.LZCOMP_L
    DECOMPRESS USING TD_SYSFNLIB.LZDECOMP_L);
```

## Related Information

- For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- See <http://zlib.net> for information about the decompression algorithm used by LZDECOMP\_L.

## TD\_LZ\_COMPRESS

Compresses any supported ALC data type or predefined type data using the Lempel-Ziv algorithm.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] TD_LZ_COMPRESS ( expression )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *expression*

- Any supported ALC data type when using the function as the compression routine for ALC on a column.
- Any supported ALC data types and all predefined data types and Distinct UDTs when calling the function.

---

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

---

## Argument Type and Rules

Expressions passed to this function must have one of the following data types:

- BYTE
- VARBYTE
- CHARACTER
- VARCHAR
- JSON
- TIME
- TIMESTAMP
- System-defined UDTs, such as ST\_Geometry and XML.
- Distinct UDTs, including LOB UDTs, ARRAY or Period data types.

Data types must match the result data type of the corresponding decompression function, TD\_LZ\_DECOMPRESS.

## Result Type

The result data types are VARBYTE or BLOB.

## Usage Notes

Numeric data types are valid when calling the function in an SQL statement, but not when using the function for ALC on a column.

If TD\_LZ\_COMPRESS is used with ALC, nulls are also compressed if those columns are nullable.

You can use the TD\_LZ\_COMPRESS function to compress JSON data; however, Teradata recommends that you use JSON\_COMPRESS instead because the JSON\_COMPRESS function is optimized for compressing JSON data.

## Uncompressing Data Compressed with TD\_LZ\_COMPRESS

To uncompress data that was compressed using TD\_LZ\_COMPRESS, use "TD\_LZ\_DECOMPRESS"..

## Related Information

- For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details about UDF implicit type conversion rules, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information about the Lempel-Ziv algorithm, see <http://zlib.net>.
- For more information about the DECAMSET function, see [DECAMSET](#).

- For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Teradata Vantage™ - Database Design*, B035-1094.
- To uncompress data that was compressed using TD\_LZ\_COMPRESS, use [TD\\_LZ\\_DECOMPRESS](#).

## TD\_LZ\_DECOMPRESS

Uncompresses data that was compressed using the TD\_LZ\_COMPRESS function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] TD_LZ_DECOMPRESS ( expression )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *expression*

Data that was compressed using the TD\_LZ\_COMPRESS function, represented as VARBYTE, BLOB, or a combination of the two.

---

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

---

## Argument Type and Rules

Expressions passed to this function must have the same data type as those of the corresponding compress function, TD\_LZ\_COMPRESS. The input should be of type VARBYTE, BLOB, or a combination of the two.

## Result Type

The result data type always matches the ALC-allowable data types.

## Usage Notes

TD\_LZ\_DECOMPRESS is normally used with ALC to uncompress table columns previously compressed with TD\_LZ\_COMPRESS.

Calling the function directly requires using the RETURNS clause to specify the desired result data type. Calling the function directly is not recommended for large data types, such as LOBs.

## Related Information

- For more information about the COMPRESS and DECOMPRESS phrases, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For an example of the use of a RETURN clause to return a specified data type, see TD\_ANYTYPE in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- See <http://zlib.net> for information about the decompression algorithm used by TD\_LZ\_DECOMPRESS.
- For more information about ALC, see the COMPRESS and DECOMPRESS phrases in *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## TS\_COMPRESS

Compresses TIME and TIMESTAMP with or without time zone to the minimum possible bytes.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] TS_COMPRESS ( td_anytype )
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *td\_anytype*

In this function, TIME or TIMESTAMP with or without time zone data types.

---

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

---

## Argument Type and Rules

The output of this function is the input to the TS\_DECOMPRESS function.

Though the input argument type is `td_anytype`, the algorithm for this function only supports `TIME` and `TIMESTAMP` data types. Each value in `TIME` and `TIMESTAMP` (for example, year, month, or day) is compressed into the minimum possible bits individually and then concatenated.

## Result Type

The result data type is `VARBYTE(20)`.

## Usage Notes

When creating algorithmic compression (ALC) columns, you should include the compression and decompression functions by which the column is going to be compressed and uncompressed respectively.

When you insert data into ALC columns, the data is compressed. When you select ALC columns the data is uncompressed.

Uncompressed Data for the Following Data Types...	Requires the Following Bytes...
TIME	6
TIMESTAMP	10
TIME WITH TIME ZONE	8
TIMESTAMP WITH TIME ZONE	12

## Examples

### Example: Compressing the TIME Data Type

Uncompressed data requires 6 bytes for the `TIME` data type, broken down into bytes, possible range and minimum possible bits as shown in the following table.

Hour	Byte	Possible Range	Minimum Possible Bits
Hour	1	00 23	5
Minute	1	00 59	6
Second	4	00 000000 61.999999	6 to 26 depending on the seconds precision

If you insert the following value for `TIME`:

```
INSERT into t1(1, TIME '03:38:06');
```

as specified in the following SQL statement:

```
CREATE TABLE table t1(pk int, col1 time(0) compress using ts_compress
decompress using ts_decompress);
```

the required bits are as follows:

- 3 hours = 5
- 38 minutes = 6
- 6 seconds = 6

The total number of required bits needed to represent TIME is 17. The minimum number of bytes is 3. Since TIME (without TIMEZONE) takes 6 bytes, TS\_COMPRESS compresses 6 bytes to 3. The function saves 3 bytes.

## Example: Compressing the TIMESTAMP Data Type

The following SQL statement creates a TIMESTAMP column in t\_timestamp that may be stored in compressed form:

```
CREATE TABLE table t_timestamp(i int, j timestamp(0) compress using
td_sysfnlib.ts_compress decompress using td_sysfnlib.ts_decompress);
```

## Related Information

- For more information about ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For more information about the COMPRESS and DECOMPRESS phrases, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## TS\_DECOMPRESS

Uncompresses TIME and TIMESTAMP data with or without time zone that the TS\_COMPRESS function compressed.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] TS_DECOMPRESS ( compressed_string )
```

## Syntax Elements

### TD\_SYSFNLIB.

Name of the database where the function is located.

### *compressed\_string*

Data that was compressed using the TS\_COMPRESS function.

---

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

---

## Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(n), where the maximum supported size (n) is 20.

The input to this function must be the output result of the TS\_COMPRESS function.

## Result Type

The result data type is *td\_anytype*.

## Example: Creating a Compressed TIME Column

The following SQL statement creates a TIME column in *t\_time* that is compressed when stored and uncompressed when retrieved from storage:

```
CREATE table t_time(i int, j time(6) compress using td_sysfnlib.ts_compress
decompress using td_sysfnlib.ts_decompress);
```

If you insert the following TIME values into column j:

```
INSERT into t_time(1, TIME'06:26:44.820000');
INSERT into t_time(2, TIME'10:26:44.820000');
```

The following SQL statement:

```
SELECT j from t_time;
```

returns the 2 uncompressed TIME values that were compressed when the values were inserted in the table *t\_time*.

## Related Information

For more information about the COMPRESS and DECOMPRESS phrases, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## TransUnicodeToUTF8

Compress the specified Unicode character data into UTF8 format.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] TransUnicodeToUTF8 ( Unicode_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *Unicode\_string*

A Unicode character string or string expression.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARCHAR(*n*) CHARACTER SET UNICODE, where the maximum supported size (*n*) is 32000. You can also pass arguments with data types that can be converted to VARCHAR(32000) CHARACTER SET UNICODE using the implicit data type conversion rules that apply to UDFs. For example, TransUnicodeToUTF8(CHAR) is allowed because it can be implicitly converted to TransUnicodeToUTF8(VARCHAR).

#### Note:

The UDF implicit type conversion rules are more restrictive than the implicit type conversion rules normally used by Vantage. If an argument cannot be converted to VARCHAR following the UDF implicit conversion rules, it must be explicitly cast.



The input to this function must be Unicode character data.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARBYTE(64000).

## Usage Notes

TransUnicodeToUTF8 compresses the specified Unicode character data into UTF8 format, and returns the compressed result. This is useful when the input data is predominantly Latin characters because UTF8 uses one byte to represent Latin characters and Unicode uses 2 bytes.

TransUnicodeToUTF8 provides good compression for Unicode strings of any length and is best used:

- On a Unicode column that contains mostly US-ASCII characters
- When the data frequently switches between:
  - Uppercase and lowercase letters
  - Digits and letters
  - Latin and non-Latin characters
- When the data is very dynamic (under frequent update)

Although you can call the function directly, TransUnicodeToUTF8 is normally used with algorithmic compression (ALC) to compress table columns. If TransUnicodeToUTF8 is used with ALC, nulls are also compressed if those columns are nullable.

## Restrictions

TransUnicodeToUTF8 can only compress character values in the 7-bit ASCII character range, from U+0000 to U+007F, also known as US-ASCII.

## Example: Uncompressing Data Compressed with TransUnicodeToUTF8

To uncompress Unicode data that was compressed using TransUnicodeToUTF8, use the TransUTF8ToUnicode function.

In this example, assume that the default server character set is UNICODE. The values of the Description column are compressed using the TransUnicodeToUTF8 function with ALC, which stores the Unicode input in UTF8 format. The TransUTF8ToUnicode function uncompresses the previously compressed values.

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPERCASE,
   Description VARCHAR(1000))
```

```
COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode);
```

## Related Information

- For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For details, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- To uncompress Unicode data that was compressed using TransUnicodeToUTF8, use the TransUTF8ToUnicode function. See [TransUnicodeToUTF8](#).
- For a detailed comparison between the Teradata-supplied compression functions and guidelines for choosing a compression function, see *Teradata Vantage™ - Database Design*, B035-1094.

## TransUTF8ToUnicode

Uncompresses the Unicode data that was compressed using the TransUnicodeToUTF8 function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[TD_SYSFNLIB.] TransUTF8ToUnicode ( compressed_string )
```

#### Note:

This function takes no arguments when used as part of the COMPRESS USING or DECOMPRESS USING phrases.

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *compressed\_string*

Unicode character data that was compressed using the TransUnicodeToUTF8 function.

## Argument Type and Rules

Expressions passed to this function must have a data type of VARBYTE(*n*), where the maximum supported size (*n*) is 64000.

The input to this function must be the output result of the TransUnicodeToUTF8 function.

If you specify NULL as input, the function returns NULL.

## Result Type

The result data type is VARCHAR(32000) CHARACTER SET UNICODE

## Usage Notes

TransUnicodeToUTF8 compresses the specified Unicode character data into UTF8 format, and returns the compressed result. This is useful when the input data is predominantly Latin characters because UTF8 uses one byte to represent Latin characters and Unicode uses 2 bytes.

TransUnicodeToUTF8 provides good compression for Unicode strings of any length and is best used:

- On a Unicode column that contains mostly US-ASCII characters
- When the data frequently switches between:
  - Uppercase and lowercase letters
  - Digits and letters
  - Latin and non-Latin characters
- When the data is very dynamic (under frequent update)

Although you can call the function directly, TransUnicodeToUTF8 is normally used with algorithmic compression (ALC) to compress table columns. If TransUnicodeToUTF8 is used with ALC, nulls are also compressed if those columns are nullable.

## Example: Uncompressing Unicode Values with TransUTF8ToUnicode

In this example, assume that the default server character set is UNICODE. The values of the Description column are compressed using the TransUnicodeToUTF8 function with ALC, which stores the Unicode input in UTF8 format. The TransUTF8ToUnicode function uncompresses the previously compressed values.

```
CREATE TABLE Pendants
  (ItemNo INTEGER,
   Gem CHAR(10) UPPERCASE,
   Description VARCHAR(1000)
    COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
    DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode);
```

## Related Information

For more information about the COMPRESS and DECOMPRESS phrases and ALC, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

# Export Width Procedures

The following sections describe the stored procedures used to:

- Create a new user-defined export width definition.
- Modify an existing user-defined export width definition.
- Delete a user-defined export width definition.

## About Export Width

The number of characters or bytes the database exports can exceed or be less than the number of characters or bytes the client application expects to receive unless the export width is set properly. The export width defines how the system reserves space for each character field in result set rows returned to Teradata clients.

You can set the DBS Control field, Export Width Table ID, to define the default export width the system uses when returning a result set to a client. For more information, see *Teradata Vantage™ - Database Utilities*, B035-1102.

You can override the system-level export width by using the EXPORTWIDTH option in the CREATE USER or MODIFY USER statements. The EXPORTWIDTH option accepts the following values:

Export Definition Name	Description
'Expected'	Provides reasonable default widths for the character data type and client form of use.
'Compatibility'	Enables Unicode data to be processed by applications that are written to process Latin or KANJI1 data.
'Maximum'	Provides maximum default width of the character data type and client form of use.
'user-defined name '	Provides user-defined default widths for the character data type and client form of use.
DEFAULT	Uses the setting of the DBS Control field: Export Width Table ID.

You can use the ReplaceExportDefinition stored procedure to:

- Create a new user-defined export definition name and the associated export width rules.
- Update the export width rules associated with an existing user-defined export definition name.

You can use the RemoveExportDefinition stored procedure to delete a user-defined export definition name and the associated export width rules.

For more information about...	See...
export widths and creating user-defined export width definitions	<i>Teradata Vantage™ - Advanced SQL Engine International Character Set Support</i> , B035-1125.
specifying a user-level export width setting	CREATE USER or MODIFY USER in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
setting the system default for the export width	DBS Control (dbscontrol) in <i>Teradata Vantage™ - Database Utilities</i> , B035-1102.

## ReplaceExportDefinition

Creates or modifies a user-defined export width definition.

### Required Privileges

You must have EXECUTE privileges on the stored procedure or on the SYSLIB database.

### Syntax

```
CALL [SYSLIB.] ReplaceExportDefinition (
  'export_definition_name',
  'export_width_rule_set',
  result_message
) [;]
```

### Syntax Elements

#### SYSLIB.

Name of the database where the function is located.

#### **export\_definition\_name**

A user-defined export definition name.

You cannot specify 'Expected', 'Compatibility', or 'Maximum'.

#### **export\_width\_rule\_set**

A user-defined BYTE string of 40 hexadecimal digits used to define the export width rules associated with *export\_definition\_name*.

#### **result\_message**

The output message of the stored procedure.

## Argument Types

Expressions passed to this procedure must have the following data types:

- *export\_definition\_name* = VARCHAR(*n*) CHARACTER SET LATIN, where the maximum value for *n* is 30.
- *export\_width\_rule\_set* = BYTE(20)
- *result\_message* = VARCHAR(310) CHARACTER SET UNICODE

You can also pass arguments with data types that can be converted to the above types using Teradata implicit data type conversion rules.

## Usage Notes

To create a new user-defined export width definition, call the ReplaceExportDefinition stored procedure with the following arguments:

- *export\_definition\_name* is a LATIN character string that specifies the name of the new export width definition.
- *export\_width\_rule\_set* is a BYTE string of 40 hexadecimal digits, which encode the export width rules for the new export width definition. For details about the export width rules, see “Export Width Rules”.
- *result\_message* is an output parameter that contains the output message of the stored procedure.

ReplaceExportDefinition creates the new export width definition with the specified rules and updates the DBC.ExportWidth table.

To modify the export width rules for an existing user-defined export width definition, call the ReplaceExportDefinition stored procedure with the following arguments:

- *export\_definition\_name* is a LATIN character string that specifies the name of an existing user-defined export width definition that you want to modify.
- *export\_width\_rule\_set* is a BYTE string of 40 hexadecimal digits, which define the new export width rules for *export\_definition\_name*. For details about the export width rules, see [Export Width Rules](#).
- *result\_message* is an output parameter that contains the output message of the stored procedure.

ReplaceExportDefinition modifies the specified export width definition with the new rules and updates the DBC.ExportWidth table. The revised export width definition takes effect at the next transaction for each affected user. In process transactions continue with the existing export width definition.

---

### Note:

You cannot specify 'Expected', 'Compatibility', or 'Maximum' for *export\_definition\_name*.

---

## Export Width Rules

*export\_width\_rule\_set* is a BYTE string of 40 hexadecimal digits with the following format:

- The 1st ten digits define the export width for LATIN strings.

- The 2nd ten digits define the export width for UNICODE strings.
- The 3rd ten digits define the export width for KANJISJIS strings.
- The 4th ten digits define the export width for GRAPHIC strings.

For example, if `export_width_rule_set` is '1112211111222232222211121111112222322222'XB, then:

- The 1st ten digits (1112211111) define the export width for LATIN strings.
- The 2nd ten digits (2222322222) define the export width for UNICODE strings.
- The 3rd ten digits (1112111111) define the export width for KANJISJIS strings.
- The 4th ten digits (2222322222) define the export width for GRAPHIC strings.

Each position in a set of 10 digits specify the export width conversion multiplier for a session character set:

This position in a set of 10 digits...	Is the conversion multiplier for....
1st	<ul style="list-style-type: none"> <li>• any session character set that ends in the string '_0I'.</li> <li>• the session character set 'KATAKANAEBDIC'.</li> </ul>
2nd	any session character set that ends in the string '_0U'.
3rd	any session character set that ends in the string '_0S'.
4th	the session character set 'UTF16'.
5th	the session character set 'UTF8'.
6th	any site-defined session character set with STATEMACHINE EUC1211.
7th	any site-defined session character set with STATEMACHINE EUC1223.
8th	any site-defined session character set with STATEMACHINE S80.
9th	any site-defined session character set with STATEMACHINE S80A1E0.
10th	any site-defined session character set with STATEMACHINE SOSIOE0F.

Each position can have a value of 1, 2, 3, or 4, except the 4th position in each 10 character set (UTF16), which must have a value of either 2 or 4.

For example, if `export_width_rule_set` is '1112211111222232222211121111112222322222'XB:

- When exporting from the LATIN server character set (1st set of 10 digits, 1112211111), for the UTF8 session character set (the 5th position in the 1st set of 10 digits), the export width is 2.
- When exporting from the UNICODE server character set (2nd set of 10 digits, 2222322222), for the UTF8 session character set (the 5th position in the 2nd set of 10 digits), the export width is 3.
- When exporting from the KANJISJIS server character set (3rd set of 10 digits, 1112111111), for the UTF16 session character set (the 4th position in the 3rd set of 10 digits), the export width is 2.

## Example

In this example, if MyExportWidth does not exist, ReplaceExportDefinition creates a new export width definition named MyExportWidth with an associated export width rule set of '1422323221332232322321123111122322323221'XB. If MyExportWidth already exists, then ReplaceExportDefinition replaces the current export width rule set of MyExportWidth with the new export width rule set of '1422323221332232322321123111122322323221'XB.

```
CALL SYSLIB.ReplaceExportDefinition
('MyExportWidth','1422323221332232322321123111122322323221'xb,msg);
```

## RemoveExportDefinition

Removes a user-defined export width definition.

### Required Privileges

You must have EXECUTE privileges on the stored procedure or on the SYSLIB database.

### Syntax

```
CALL [SYSLIB.] RemoveExportDefinition (
  'export_definition_name',
  result_message
) [;]
```

### Syntax Elements

#### SYSLIB.

Name of the database where the function is located.

#### *export\_definition\_name*

A user-defined export definition name.

You cannot specify 'Expected', 'Compatibility', or 'Maximum'.

#### *result\_message*

The output message of the stored procedure.

## Argument Types

Expressions passed to this procedure must have the following data types:



- *export\_definition\_name* = VARCHAR(*n*) CHARACTER SET LATIN, where the maximum value for *n* is 30.
- *result\_message* = VARCHAR(100) CHARACTER SET UNICODE

You can also pass arguments with data types that can be converted to the above types using Teradata implicit data type conversion rules.

## Usage Notes

To delete an existing user-defined export width definition, call the RemoveExportDefinition stored procedure with the following arguments:

- *export\_definition\_name* is a LATIN character string that specifies the name of the existing export width definition you want to delete.
- *result\_message* is an output parameter that contains the output message of the stored procedure.

RemoveExportDefinition deletes the specified export width definition from the DBC.ExportWidth table.

---

### Note:

You cannot specify 'Expected', 'Compatibility', or 'Maximum' for *export\_definition\_name*. In addition, you cannot delete an export width definition if it is assigned to a user.

---

## Example

The following statement deletes the user-defined export width definition named MyExportWidth.

```
CALL SYSLIB.RemoveExportDefinition('MyExportWidth',msg);
```

# File System Information Macros and Functions

These macros and functions allow you to use various Ferret SHOW commands to display Teradata File System statistics about data block size, number of rows per data block, and information about the compression status of data blocks for specified tables or groups of tables. You can also run frequency of access reports for database objects on a specific AMP.

The information is similar to that displayed by the SHOWBLOCKS, SHOWCOMPRESS, and SHOWWHERE commands of the Ferret utility. Because the information from these macros and functions is created in regular database tables, it can be easier to share and process the information. Permissions to create and view this information are also easier to manage using standard SQL GRANT and REVOKE privilege statements.

For more information on Ferret, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## CreateFsysInfoTable/CreateFsysInfoTable\_ANSI

Creates a database table to hold file system information generated by PopulateFsysInfoTable macro.

---

### Note:

If you use ANSI session mode, use the \_ANSI form of the macro.

---

The PopulateFsysInfoTable or PopulateFsysInfoTable\_ANSI macro can be called with empty strings for the database and table names, which redirects the output to the screen.

### Required Privileges

In addition to the privileges already mentioned, to run this macro you must have EXECUTE privileges on the CreateFsysInfoTable macro or on the database containing the macro.

The macro name and arguments are case-insensitive.

### Syntax

```
[DBC.] { CreateFsysInfoTable | CreateFsysInfoTable_ANSI } (
  'target_database_name',
  'target_table_name',
  'target_table_type',
  'fallback',
  { 'SHOWBLOCKS' | 'SHOWWHERE' | 'SHOWCOMPRESS' },
  'display_option',
  'mapname'
)
```

## Syntax Elements

### *target\_database\_name*

The database in which the table will be created.

---

**Note:**

You must have appropriate privileges to create tables in the target database.

---

### *target\_table\_name*

The name of the table that will be created to hold file system information.

---

**Note:**

You must have appropriate privileges to create tables in the target database.

---

### *target\_table\_type*

Specifies whether the table is a permanent, global temporary, or volatile table.

Valid values:

- PERM
- GLOBALTEMP
- VOLATILE

### *is\_fallback*

Specifies whether the database maintains a fallback copy of the table.

Valid values:

- Y for yes, meaning the table will have fallback
- N for no, meaning the table will not have fallback

## SHOWBLOCKS

## SHOWCOMPRESS

## SHOWWHERE

The Ferret SHOW command that this output will resemble.

The listed SHOW\* command includes a MapName column. The MapName column appears in the listed SHOW\* command displays, and is usually passed the default Teradata map name.

### *display\_option*

The level of file system detail that can be stored in the table.

Valid values:

- S provides minimal amount of file system information
- M provides a medium amount of file system information
- L provides the maximum amount of file system information

For more information about the specific information displayed, see [PopulateFsysInfoTable/PopulateFsysInfoTable\\_ANSI](#).

---

**Note:**

The display option chosen for CreateFsysInfoTable must match the display option to be used for the corresponding execution of the PopulateFsysInfoTable macro.

If output is directed to the screen (for example, empty strings are specified for database and table names), then table creation is not required using CreateFsysInfoTable or CreateFsysInfoTable\_ANSI macro call. Any display option can be used.

---

***mapname***

Supports table creation in the specified map. If you do not want to choose a map name for the target table creation, you can specify empty strings ( ' ') for this parameter. In that case, the table is created in the default map.

## Examples

### Example: Create a File System Information Table for Short Display for SHOWBLOCKS

This statement creates a volatile, non-fallback table to hold a minimal display of disk block information.

```
exec dbc.createfsysinfotable ('SystemInfo'
,'datablockinfo','volatile','n','showblocks','s', 'TD_Map1');
```

### Example: Create a File System Information Table for Short Display for SHOWCOMPRESS

This statement creates a PERM type of target table to hold the SHOWCOMPRESS short display output.

```
exec dbc.createfsysinfotable ('targetdb',
'targettable','perm','y','showcompress','s', 'TD_Map1');
```

The 'M' display option is not supported for SHOWCOMPRESS. Use the 'L' option for long output.

## Example: Create a File System Information Table for Short Display for SHOWWHERE

This statement creates a perm, non-fallback target table 'showw\_s' to hold the SHOWWHERE short display output:

```
exec dbc.createfsysinfotable ('SystemInfo'
', 'showw_s', 'perm', 'n', 'showwhere', 's', 'TD_Map1');
```

Change the option to 'M' or 'L' to see medium or long outputs.

## PopulateFsysInfoTable/PopulateFsysInfoTable\_ANSI

Generates file system data to populate a table created by the CreateFsysInfoTable macro.

### Note:

If you use ANSI session mode, use the \_ANSI form of the macro.

### Required Privileges

In addition to the privileges mentioned above, to run this macro you must have EXECUTE privileges on the PopulateFsysInfoTable macro or on the database containing the macro.

The macro name and arguments are case-insensitive.

### Syntax

```
[DBC.] { PopulateFsysInfoTable | PopulateFsysInfoTable_ANSI } (
  { 'input_database_name', 'input_table_name', |
    'DBC', 'input_class'
  }
  { 'SHOWBLOCKS' | 'SHOWWHERE' | 'SHOWCOMPRESS' },
  'display_opt',
  'target_database_name',
  'target_table_name'
)
```

### Syntax Elements

#### *input\_database\_name*

The database containing the tables for which file system information will be generated, depending on the command SHOWBLOCKS, SHOWCOMPRESS, or SHOWWHERE.

**Note:**

You must have appropriate privileges to select data from the tables in the input database.

If you specify database DBC, you can use the *input\_class* argument to select a class of tables for which information will be generated.

***input\_table\_name***

The name of the table for which file system information will be generated.

**Note:**

You must have appropriate privileges to select data from the tables in the input database.

***input\_class***

A class of tables for which file system information will be generated.

You can specify an *input\_class* only when *input\_database\_name* is DBC.

Valid values for *input\_class*:

- CLASSPERM
- CLASSJRNL
- CLASSREDRIVE
- CLASSGLOBALTEMP
- CLASSSPOOL
- CLASSWAL
- CLASSALL (shows all of the above-mentioned classes of tables)

**Note:**

*input\_class* is only supported for SHOWWHERE. The *input\_class* option is not available if the macro is used to generate SHOWBLOCKS information, due to high resource utilization.

**SHOWBLOCKS**

Displays statistics about data block size, number of rows per data block, and information about the compression status of data blocks and tables. This output resembles Ferret SHOWBLOCKS command output.

*display\_opt* is the level of file system detail generated and populated into the target table. The display option you specify for PopulateFsysInfoTable must match the display option used for the corresponding execution of the CreateFsysInfoTable macro.

- **S**

Includes the following information for the primary data subtable of the specified tables:

- A histogram of block sizes.
- The minimum, average, and maximum block size per subtable.
- Block compression information (status, estimated compression ratio, estimated uncompressed).

#### Description of SHOWBLOCKS SQL Output Columns for S Display

For more information about the output of this macro, see *SQL SHOWBLOCKS and SQL SHOWWHERE Orange Book*, 541-0010699.

---

#### Note:

The column names of the output when redirected to a target table versus when results are displayed, are not the same; see the Orange Book for details.

---

- **TableID:** The TableID column in the SHOWBLOCK SQL output shows Unique0, Unique1 in byte-flipped format, and zeros for TypeAndIndex. This matches the common Teradata TableID format, so joins of target tables to dictionary tables can be performed easily. The TableID can also be compared to other places where TableIDs are printed (checktable, log messages, etc.).
- **TableIDTAI:** Shows the TypeAndIndex part of the TableID. In the 'S' display output, it is always 1024 (= 0x400, the primary subtable's ID).
- **CompressionMethod:** One of the table level attributes (TLA) is Block Level Compression (BLC) and this attribute can be used to allow or disallow BLC on a table. The CompressionMethod column shows the BLC TLA of the table for which the SHOWBLOCKS information is being requested. A table with BLC TLA of AUTOTEMP has AUTOTEMP for this column. If the BLC TLA is DEFAULT, then this column shows the system default BLC TLA at the time when the macro is invoked.
- **CompressionState:** The CompressionState of a table can be either C (Fully Compressed), PC (Partially Compressed), U (Fully Uncompressed), or N (Not Compressible).
  - If the table is not eligible for compression, CompressionState is set to N for that table. For example, DBC dictionary tables, whose unique part of the TableID is less than 1000, will never get compressed; so, they will have N in the compression status column.
  - When Data Blocks are created they are represented in the File System index structure by Data Block Descriptor entries (DBDs). When DBDs are created for a table eligible for compression, the COMPRESSIBLE flag is set in the DBD. The flag implies that the file system tried to compress the corresponding Data Block and that subsequent modifications of the Data Block should again try and compress the result. There DBSCONTROL tunables that

affect whether the Data Block is actually stored in compressed format are MinDBSectsToCompress and MinPercentCompReduction.

- If all DBDs of the primary subtable of the table have the COMPRESSIBLE flag set, then the CompressionState is 'C'. If the table is eligible for compression, but some DBDs have the COMPRESSIBLE flag set and some do not, then the compression status is set to 'PC' to indicate that the table is partially compressed. This can happen in the following situations:
    - An aborted FERRET COMPRESS or UNCOMPRESS operation.
    - A populated table whose BLOCKCOMPRESSION attribute was changed and no IMMEDIATE clause was given.
    - A partitioned primary index table where some partitions are heavily used and some are not.
  - If the table is eligible for compression, but none of the primary subtable DBDs have the COMPRESSIBLE flag set, then the compression status for that table is marked as 'U'.
- EstCompRatio: This column shows the Estimated Compression Ratio for the table for which SHOWBLOCKS information is being extracted. For each available sector-size range of the Data Blocks, a sample of three Data Blocks is read from the Primary subtable blocks. In the compressed Data Block header, both compressed length and uncompressed lengths are recorded, so the compression ratio can easily be obtained using the following computation without having to do the trial compression.
 
$$\text{Compression ratio} = \{1 - [\text{size of sectors after compression} / \text{original uncompressed sector size}] \} \times 100$$
  - EstPctOfUncompDBs: In this column SQL SHOWBLOCKS indicates the estimated percentage of Blocks uncompressed of the table. For subtables marked with a 'C', the first three Data Blocks of each sector-size range are sampled and checked against their DBD's compression state/info. If these blocks are actually compressed then the estimated value for the corresponding column is displayed as 0%.
  - PctDBsIn1to8 (and other similar PctDBsInXtoY columns): This column shows the percent of total Data Blocks whose size is in the range of 1 to 8 sectors (both inclusive). Likewise for the rest of the PctDBsInXtoY columns.
  - MinDBSize, AvgDBSize, and MaxDBSize: These columns respectively show the minimum size of the Data Blocks (in sectors), average size of the Data Blocks (in sectors) and maximum size of the data blocks (in sectors) for the specified table. The values are computed in each AMP as the cylinder indexes (CIs) are read and then aggregated across all AMPs.
  - TotalNumDBs: This value represents the total number of Data Blocks comprising the subtable identified by TableIDTAI. This value is counted in each AMP as the CIs are read and then aggregated across all AMPs.
  - LrgCyls, SmlCyls:



- **LrgCyls** – The number of cylinders, in units of Large cylinders, that the table identified by the TableIDTAI occupies on the disk.
- **SmlCyls** – The number of cylinders, in units of Small cylinders, that the table identified by the TableIDTAI occupies on the disk.

This value is counted in each AMP as the CIs are read and then aggregated across all AMPs. For a given row, only one of the columns is filled in. Note, a cylinder can contain multiple subtables and multiple tables. Therefore, adding up the cylinder counts from each row of the 'S' display can exceed the actual total number of cylinders used.

- **M**

Includes the same information as S, but display the table name, map that the table uses for data distribution, and statistics for all subtables. For each subtable, display the BLOCKCOMPRESSION value set when the table was created or altered.

#### **Description of SHOWBLOCKS Output Columns for M Display**

The main difference between columns produced by the 'S' option versus the 'M' option is that 'S' produces block histogram details for only the primary subtable whereas 'M' produces the block histogram details for each subtable of the specified input. Additionally, for partially compressed subtables instead of a PC row, there are two separate C and U rows.

Column values other than CompressionState are produced in the same fashion as for the 'S' option, for each of the rows. The decision to display either a C or U row is made on a subtable basis. For example, with a fallback table you could get one row for the primary subtable (assuming the primary is not compressed) and both a C row and a U row for fallback subtable if the fallback subtable is partially compressed.

Note, a cylinder can contain multiple subtables and multiple tables. Therefore, adding up the cylinder counts from each row of the 'M' display can exceed the actual total number of cylinders used.

- **L**

Includes the following information for each block size category of each subtable of the specified tables:

- Table name
- Map that the table uses for data distribution
- Number of blocks
- The minimum, average, and maximum number of rows per data block size
- Block compression information (status, estimated compression ratio, estimated percent uncompressed)
- Statistics

#### **Description of SHOWBLOCKS SQL Output Columns for L Display**

In addition to the Compression related columns that the 'M' option produces, the 'L' option produces the DBSize, DBsPerSize, PctOfDBsInSubTable, MinNumRowsPerDB, AvgNumRowsPerDB, and MaxNumRowsPerDB columns for each subtable of the specified input. Because a row is generated for each different DB size, the columns referring to DB size ranges are not required and are not part of the L display.

**Zero DBSize row:** In long display format, the total cylinder count columns LrgCyls and SmlCyls cannot be filled in for each output row since the number of Data Blocks (DBsPerSize) that exist for a given DBSize may not occupy a complete cylinder. Representing fractional parts in the cylinder count is not done. Also, if either LrgCyls or SmlCyls columns were filled in with a rounded off number of cylinders for each DBSize row, the sum total of all cylinders for a given subtable would be magnified way out of proportion. To avoid this issue, for each subtable in the long display, a special (extra) row with a DBSize of 0 is produced. In that particular row, after filling in the basic column values (for example, TheDate, TheTime, DatabaseName, TableName, TableID, TableIDTAI, and CompressionMethod), the cylinder count columns (LrgCyls or SmlCyls) are set to the total number of cylinders that the subtable occupies. All other columns in the row are 0 or blank depending on their data type. Note, Vantage does not create Data Blocks with a size of 0 sectors.

## SHOWCOMPRESS

Displays compression information for tables on the system. This output resembles the Ferret SHOWCOMPRESS command output.

*display\_opt* is the level of display detail generated. The display option you specify for PopulateFsysInfoTable must match the display option used for the corresponding execution of the CreateFsysInfoTable macro.

- **S**

The display option you specify for PopulateFsysInfoTable must match the display option used for the corresponding execution of the CreateFsysInfoTable macro. This is the default display.

### Description of SHOWCOMPRESS SQL Output Columns for S Display

For more information about the output of this macro, see *SQL SHOWBLOCKS and SQL SHOWWHERE Orange Book*, 541-0010699.

---

#### Note:

The column names of the output when redirected to a target table versus when results are displayed, are not the same; see the Orange Book for details.

- **TableID:** The TableID column in the SHOWCOMPRESS SQL output shows Unique0, Unique1 in byte-flipped format, and zeros for TypeAndIndex. This matches the common Teradata TableID format. Therefore, joins of target tables to dictionary

tables can be performed easily. The TableID can also be compared to other places where TableIDs are printed (checktable, log messages, etc.).

- TableIDTAI: Shows the TypeAndIndex part of the TableID.
  - TLA\_BLCOption: This column shows the table level attribute (TLA) of Block Level Compression Option (BLC Option). It shows the compression method of the table for which the SHOWCOMPRESS information was requested, such as AUTOTEMP, MANUAL, ALWAYS, or NEVER.
  - System\_BLCOption: If the TLA BLC Option is DEFAULT, this column shows the system default BLC Option as AUTOTEMP, MANUAL, ALWAYS, or NEVER, at the time when the macro is invoked.
  - TLA\_BLCALG: This column shows the table level attribute (TLA) of Block Level Compression Algorithm (BLC ALG) of the table for which the SHOWCOMPRESS information is being requested. The supported algorithms are ZLIB, IPPZLIB, and ELZS\_H.
  - System\_BLCALG: If the TLA BLC Algorithm is DEFAULT, then this column shows the system default BLC algorithm value as ZLIB, IPPZLIB, or ELZS\_H, at the time when the macro is invoked.
  - TLA\_BLCLevel: This column shows the table level attribute (TLA) of Block Level Compression Level (BLC Level) of the table for which the SHOWCOMPRESS information is being requested. It is meaningful only when the compression algorithm is ZLIB or IPPZLIB. It indicates the compression level from 1 to 9.
  - System\_BLCLevel: If the TLA BLC Level is DEFAULT, then this column shows the system default BLC Level value at the time when the macro is invoked.
  - MapName: This column shows the name of the map used for the specified table.
- **L**

The display option you specify for PopulateFsysInfoTable must match the display option used for the corresponding execution of the CreateFsysInfoTable macro.

---

**Note:**

This display option requires longer to display than the /S option because all of the data blocks of the tables in scope must be examined.

---

The output shows blocks that are compressed, uncompressed, and disqualified. Disqualified blocks are those that did not qualify for compression for one of these reasons:

- Some data blocks fall below BLC compression threshold levels, which are set in the Compression group of DBS Control fields.
- Some data blocks are too small to be compressed.
- Some special subtables are never compressed.

**Description of SHOWCOMPRESS SQL Output Columns for L Display**

In addition to the Compression related columns that the 'S' option produces, the 'L' option produces the CompressionState, CompressionAlgorithm, CompressionLevel, ExactCompRatio, ExactPctofBlocks, ExactPctofData, and ExactUsedGB columns for each subtable of the specified input.

- **CompressionState:** This column provides the compression state of each subtable of the specified table. Possible values are COMPRESSED (DBs of the subtable are compressed), DISQUALIFIED (DBs of the subtable are compressible but not compressed) and UNCOMPRESSED (DBs of the subtable are neither compressible nor compressed). The information is summarized and provided at the table-level as well.
- **CompressionAlgorithm:** This information is obtained by traversing each data block of subtable or table as there may be a possibility of multiple compression algorithms being used for a single subtable or table. Possible compression algorithms are ZLIB, ELZS\_H, IPPZLIB, and N/A (when compression state is either compressible or uncompressed). This information is displayed at the table and subtable level.
- **CompressionLevel:** This information is obtained by traversing every data block of the subtable or table as there may be a possibility of multiple compression levels for the same or different compression algorithm being used for a single subtable or table. In case of the same compression algorithm with a different compression level for a subtable or table, the range of values will be displayed. Possible compression levels are from 1 to 9 (when compression algorithm used is ZLIB or IPPZLIB) and N/A (otherwise). This information is displayed at the table and subtable level.
- **ExactCompRatio:** This column shows the Exact Compression Ratio for the table for which SHOWCOMPRESS information is being extracted. This information is calculated after traversing each of the data blocks present in the compressed table. This information is presented at both the table and subtable level. It is displayed as a percentage.
- **ExactPctofBlocks:** In this column, SQL SHOWCOMPRESS indicates the exact percentage of blocks that are in the specified Compression State, Algorithm, and Level. It is calculated after traversing each of the data blocks and checking whether it is compressed, compressible, or uncompressed. This information is presented at both the table and subtable level. It is displayed as a percentage.
- **ExactPctofData:** This column provides the exact percentage of the total data that is either compressed using a particular algorithm, compressible, or uncompressed. It is displayed as a percentage.
- **ExactUsedGB:** This column provides the exact used size for each compression algorithm (ZLIB, ELZS\_H, and IPPZLIB) or exact used size for compressible or uncompressed data blocks in gigabytes. Summation of all the values within the subtable of a table for a particular compression algorithm/compressible/uncompressed should be equal to the value at the table level for the corresponding compression algorithm/compressible/uncompressed.

**SHOWWHERE**

Displays information about cylinder allocation and temperature. This output resembles the Ferret SHOWWHERE command output.

*display\_opt* is the level of file system detail generated and populated into the target table. The display option you specify for PopulateFsysInfoTable must match the display option used for the corresponding execution of the CreateFsysInfoTable macro.

- **S**

Displays a summary listing of the cylinders showing one line for every cylinder type.

- **M**

Displays a medium length listing of the cylinders with one line for every cylinder type per AMP (vproc).

- **L**

Displays a long listing of the cylinders with one line for every cylinder type per AMP (vproc) per storage device.

**Description of SHOWWHERE SQL Output Columns**

For more information about the output of this macro, see *SQL SHOWBLOCKS and SQL SHOWWHERE Orange Book*, 541-0010699.

**Note:**

The column names of the output when redirected to a target table versus when results are displayed, are not the same; see the Orange Book for details.

- **TableID**: Similar to SHOWBLOCKS SQL output, the TableID column in the SHOWWHERE SQL output shows the Unique0, Unique1 values in byte-flipped format.
- **TableIDTAI**: Currently there is no display option which produces SHOWWHERE rows for each subtable. The SHOWWHERE processing is done for the entire table. Therefore, the TableIDTAI column data is filled with zeros when the output is directed to the target table. The column has blanks in it when the output is directed for immediate results (to your screen, for example), such as if the target database/target table are not used when executing the PopulateFsysInfoTable macro.
- **Vproc**: The content of this column depends on the display options specified.
  - 'S' – Because only a system-wide result is shown for each table or class of tables, a VPROC number is inappropriate, so -1 is assigned to Vproc.
  - 'M' and 'L' – The vproc number is assigned to Vproc.
- **Drive**: The content of this column depends on the display options specified.
  - 'S' – Because only a system-wide result is shown for each table or class of tables, a DRIVE number is inappropriate, so DRIVE is -1 in this case.

- 'M' – Because only the AMP-wide result is shown for each table or class of tables, a DRIVE number is inappropriate, so DRIVE is -1 in this case.
- 'L' – The DRIVE number is displayed. Note, there may be more lines in the 'L' display if multiple drives/AMP are configured.
- LrgCyls, SmlCyls:
  - LrgCyls – The number of cylinders, in units of Large cylinders, that the table identified by the TableID occupies on the disk for the given Grade.
  - SmlCyls – The number of cylinders, in units of Small cylinders, that the table identified by the TableID occupies on the disk for the given Grade.

For a given row, only one of the columns is filled in.

- TableType: Indicates the type of the source table for which the SHOWWHERE output rows are produced. Although WAL is not a table type, its cylinders are allocated separately from table cylinders, so they are reported separately.
- Grade: TVS categorizes all cylinders into one of the three grades: FAST, MEDIUM or SLOW. For a given table, SHOWWHERE produces one row for each of these grades occupied by cylinders of the targets being displayed.
- GradePct: GradePct represents the percent of cylinders of the table which are currently in the particular grade as indicated by the Grade column. It is computed based on the following formula:

$$\frac{\text{Total Cylinder of the table in the grade identified by the current row}}{\text{Total Cylinder in the table}}$$

- VHCyls: VHCyls represents the number of cylinders which are of the VERYHOT (VH) category.
- HotCyls, WarmCyls, and ColdCyls: These are the number of HOT/WARM/COLD cylinders respectively. Depending upon the frequency of access to the cylinders (the frequency of reads/writes to the cylinders), TVS determines the temperature for the cylinders and accordingly classifies them as HOT, WARM, or COLD.
- CylsinSSD: This column represents the number of cylinders that reside on the Solid State Device. There can be a lag between when the temperature category of a cylinder changes and when that cylinder is moved into or out of SSD. So, the corresponding temperature might be older than expected. As of Teradata Database 15.10, this field is not yet populated. It will have zeros when the output of the PopulateFsysInfoTable macro is directed to a target table and will have blanks when the output is directed for immediate results, such as if the target database/target table are not used when executing the macro.
- TemperaturesMature: The temperature and grade information is only meaningful if temperatures on the system are mature, such that the system has been in use long enough to gather meaningful statistics. A 'Y' in this column indicates temperatures are mature and an 'N' indicates they are not mature. If temperatures are not mature, users

are advised to use the system awhile longer before making use of the results of the SHOWWHERE SQL.

### ***target\_database\_name***

The database in which the target table was created by the CreateFsysInfoTable macro.

If this argument and *target\_table\_name* are empty strings, the output will be directed to the computer screen.

---

#### **Note:**

You must have appropriate privileges to create or insert into tables in the target database.

---

Calling the CreateFsysInfoTable CreateFsysInfoTable\_ANSI is unnecessary if the macro is called with empty strings for the database and table names

### ***target\_table\_name***

The name of the table which was created by the CreateFsysInfoTable macro in the target database to hold file system information.

If this argument and *target\_database\_name* are empty strings, the output will be directed to the computer screen.

---

#### **Note:**

You must have appropriate privileges to insert into *target\_table\_name* in the target database. If you created *target\_table\_name* in the target database, then you already have implicit INSERT privileges on the target table.

Calling the CreateFsysInfoTable CreateFsysInfoTable\_ANSI is unnecessary if the macro is called with empty strings for the database and table names.

---

## **Examples**

### **Example: Populate a File System Information Table for SHOWBLOCKS Output, Option 'S'**

As a first step, you can create a target table to store the file system information rows in.

Create the target table manually or by using the CreateFsysInfoTable macro:

```
exec
createfsysinfotable('Call_Logs','showbtrgttableshort','perm','y','showblocks','s
','TD_Map1');
```

The following is the definition of the target table for S display option when creating the target table for storing the file system information for SHOWBLOCKS:

```
show table Call_Logs.showbtrgttable;

CREATE SET TABLE Call_Logs.showbtrgttable,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO,
  (
    TheDate DATE FORMAT 'YY/MM/DD',
    TheTime TIME(6),
    DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TableID BYTE(6),
    TableIDTAI INTEGER,
    CompressionMethod CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
    CompressionState VARCHAR(2) CHARACTER SET LATIN NOT CASESPECIFIC,
    EstCompRatio DECIMAL(5,2),
    EstPctOfUncompDBs DECIMAL(5,2),
    PctDBsIn1to8 BYTEINT,
    PctDBsIn9to24 BYTEINT,
    PctDBsIn25to64 BYTEINT,
    PctDBsIn65to120 BYTEINT,
    PctDBsIn121to168 BYTEINT,
    PctDBsIn169to216 BYTEINT,
    PctDBsIn217to256 BYTEINT,
    PctDBsIn257to360 BYTEINT,
    PctDBsIn361to456 BYTEINT,
    PctDBsIn457to512 BYTEINT,
    PctDBsIn513to760 BYTEINT,
    PctDBsIn761to1024 BYTEINT,
    PctDBsIn1025to1034 BYTEINT,
    PctDBsIn1035to1632 BYTEINT,
    PctDBsIn1633to2048 BYTEINT,
    MinDBSize INTEGER,
    AvgDBSize INTEGER,
    MaxDBSize INTEGER,
```



```

    TotalNumDBs BIGINT,
    LrgCyls BIGINT,
    SmlCyls BIGINT,
    MapName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL
)
PRIMARY INDEX ( TheDate ,TheTime ,TableID ,CompressionState );

```

The next example shows the BTEQ output from running `PopulateFsysInfoTable` macro with the S display option, and empty strings for *target\_database\_name* and *target\_table\_name* to force the output to display on screen.

The column titles names of the output change when redirected to the screen instead of using the target table to store the output. For example, the column name is “PctDBsIn1to8” in the target table and the corresponding column is “% Of DBs In 1 to 8” when the output is redirected to the screen.

The warning that is returned after `PopulateFsysInfoTable` macro is expected behavior when the output is displayed to the screen.

---

**Note:**

The BTEQ `.foldline` and `.sidetitles` commands have been used to cause the column titles of the returned data set to be displayed on the side, and the corresponding values of those columns to be shown beside the column titles.

---

**Note:**

The following PopulateFsysInfoTable() call example redirects the output/result to the screen (empty strings specified for database and table names). If inserting the output/result to a specific table, ensure the table was created with the CreateFsysInfoTable() macro and that the same /display\_opt is specified in both the PopulateFsysInfoTable and CreateFsysInfoTable macro calls.

```
exec
populatefsysinfotable('Call_Logs', 'Daily_Log','showblocks','s','','');
*** Procedure has been executed.
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 34 seconds.

*** ResultSet# 1 : 1 rows returned by "SYSLIB.POPULATEFSYSINFOTABLESP".
      Date 13/12/22
      Time 21:45:29
      DB//Name Call_Logs
      Tbl//Name Daily_Log
      TID                0000E90A0000
      TID//TAI           1024
      Comp//Method MANUAL
      Comp//State U
      Est Comp//Ratio(%)
      Est Uncomp//DBs(%)
      % Of DBs In//1 to 8//Sects
      % Of DBs In//9 to 24//Sects
      % Of DBs In//25 to 64//Sects
      % Of DBs In//65 to 120//Sects 50
      % Of DBs In//121 to 168//Sects
      % Of DBs In//169 to 216//Sects
      % Of DBs In//217 to 256//Sects 50
      % Of DBs In//257 to 360//Sects
      % Of DBs In//361 to 456//Sects
      % Of DBs In//457 to 512//Sects
      % Of DBs In//513 to 760//Sects
      % Of DBs In//761 to 1024//Sects
      % Of DBs In//1025 to 1304//Sects
      % Of DBs In//1305 to 1632//Sects
      % Of DBs In//1633 to 2048//Sects
      Min DB Size        96
      Avg DB Size        176
      Max DB Size        254
      Total DBs           8
      Lrg Cyls            4
      Sml Cyls
      Map//Name
```

## Example: Populate a File System Information Table for SHOWBLOCKS Output, Option 'M'

As a first step, you can create a target table to store the file system information rows in.

Create the target table manually or by using the CreateFsysInfoTable macro:

```
exec createfsysinfotable('Call_Logs','showbtrgttablemedium','perm','y',
'showblocks','m','TD_Map1');
```

The following is the definition of the target table for M display option when creating the target table for storing the file system information for SHOWBLOCKS:

```
show table Call_Logs.showbtrgttablemedium;
CREATE SET TABLE Call_Logs.showbtrgttablemedium ,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
    (
        TheDate DATE FORMAT 'YY/MM/DD',
        TheTime TIME(6),
        DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC
NOT NULL,
        TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT
NULL,
        TableID BYTE(6),
        TableIDTAI INTEGER,
        CompressionMethod CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
        CompressionState VARCHAR(2) CHARACTER SET LATIN NOT CASESPECIFIC,
        EstCompRatio DECIMAL(5,2),
        EstPctOfUncompDBs DECIMAL(5,2),
        PctDBsIn1to8 BYTEINT,
        PctDBsIn9to24 BYTEINT,
        PctDBsIn25to64 BYTEINT,
        PctDBsIn65to120 BYTEINT,
        PctDBsIn121to168 BYTEINT,
        PctDBsIn169to216 BYTEINT,
        PctDBsIn217to256 BYTEINT,
        PctDBsIn257to360 BYTEINT,
        PctDBsIn361to456 BYTEINT,
        PctDBsIn457to512 BYTEINT,
        PctDBsIn513to760 BYTEINT,
```

```

PctDBsIn761to1024 BYTEINT,
PctDBsIn1025to1304 BYTEINT,
PctDBsIn1305to1632 BYTEINT,
PctDBsIn1633to2048 BYTEINT,
MinDBSize INTEGER,
AvgDBSize INTEGER,
MaxDBSize INTEGER,
TotalNumDBs BIGINT,
LrgCyls BIGINT,
SmlCyls BIGINT)
MapName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL
PRIMARY INDEX (TheDate, TheTime, TableID ,TableIDTAI ,CompressionState);
BTEQ -- Enter your SQL request or BTEQ command:

```

The next example shows the BTEQ output from running `PopulateFsysInfoTable` macro with the `M` display option, and empty strings for `target_database_name` and `target_table_name` to force the output to display on screen.

The column titles names of the output change when redirected to the screen instead of using the target table to store the output. For example, the column name is “PctDBsIn1to8” in the target table and the corresponding column is “% Of DBs In 1 to 8” when the output is redirected to the screen.

The warning that is returned after `PopulateFsysInfoTable` macro is expected behavior when the output is displayed to the screen.

---

#### Note:

The BTEQ `.foldline` and `.sidetitles` commands have been used to cause the column titles of the returned data set to be displayed on the side, and the corresponding values of those columns to be shown beside the column titles. This example shows two rows of output. The first column of each row is the Date column.

---

#### Note:

The following `PopulateFsysInfoTable()` call example redirects the output/result to the screen (empty strings specified for database and table names). If inserting the output/result to a specific table, ensure the table was created with the `CreateFsysInfoTable()` macro and that the same `/display_opt` is specified in both the `PopulateFsysInfoTable` and `CreateFsysInfoTable` macro calls.

---

```

exec populatefsysinfotable('Call_Logs', 'Daily_Log','showblocks','m','','');
*** Procedure has been executed.
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 32 seconds.

*** ResultSet# 1 : 2 rows returned by "SYSLIB.POPULATEFSYSINFOTABLESP".
        Date 13/12/22

```

```

Time 21:46:12
DB//Name Call_Logs
Tbl//Name Daily_Log
TID                0000E90A0000
TID//TAI           0
Comp//Method MANUAL
Comp//State N
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
  % Of DBs In//1 to 8//Sects 100
  % Of DBs In//9 to 24//Sects
  % Of DBs In//25 to 64//Sects
  % Of DBs In//65 to 120//Sects
  % Of DBs In//121 to 168//Sects
  % Of DBs In//169 to 216//Sects
  % Of DBs In//217 to 256//Sects
  % Of DBs In//257 to 360//Sects
  % Of DBs In//361 to 456//Sects
  % Of DBs In// 457 to 512//Sects
  % Of DBs In//513 to 760//Sects
  % Of DBs In//761 to 1024//Sects
  % Of DBs In//1025 to 1304//Sects
  % Of DBs In//1305 to 1632//Sects
  % Of DBs In//1633 to 2048//Sects
Min DB Size        2
Avg DB Size        2
Max DB Size        2
Total DBs          4
Lrg Cyls           4
Sml Cyls
Map//Name
Date 13/12/22
Time 21:46:12
DB//Name Call_Logs
Tbl//Name Daily_Log
TID                0000E90A0000
TID//TAI           1024
Comp//Method MANUAL
Comp//State U
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
  % Of DBs In//1 to 8//Sects
  % Of DBs In//9 to 24//Sects
  % Of DBs In//25 to 64//Sects

```

```

% Of DBs In//65 to 120//Sects 50
% Of DBs In//121 to 168//Sects
% Of DBs In//169 to 216//Sects
% Of DBs In//217 to 256//Sects 50
% Of DBs In//257 to 360//Sects
% Of DBs In//361 to 456//Sects
% Of DBs In// 457 to 512//Sects
% Of DBs In//513 to 760//Sects
% Of DBs In//761 to 1024//Sects
% Of DBs In//1025 to 1304//Sects
% Of DBs In//1305 to 1632//Sects
% Of DBs In//1633 to 2048//Sects
Min DB Size          96
Avg DB Size          176
Max DB Size          254
Total DBs            8
Lrg Cyls             4
Sml Cyls
Map//Name

```

## Example: Populate a File System Information Table for SHOWBLOCKS Output, Option 'L'

As a first step, you can create a target table to store the file system information rows in.

Create the target table manually or by using the CreateFsysInfoTable macro:

```

exec
createfsysinfotable('Call_Logs','showbtrgttablelong','perm','y','showblocks','l'
, 'TD_Map1');

```

The following is the definition of the target table for L display option when creating the target table for storing the file system information for SHOWBLOCKS:

```

show table Call_Logs.showbtrgttablelong;
CREATE SET TABLE Call_Logs.showbtrgttablelong,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
(
    TheDate DATE FORMAT 'YY/MM/DD',
    TheTime TIME(6),

```

```

        DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC
NOTNULL,
        TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
        TableID BYTE(6),
        TableIDTAI INTEGER,
        CompressionMethod CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
        CompressionState VARCHAR(2) CHARACTER SET LATIN NOT CASESPECIFIC,
        EstCompRatio DECIMAL(5,2),
        EstPctOfUncompDBs DECIMAL(5,2),
        DBSize INTEGER,
        DBsPerSize BIGINT,
        PctOfDBsInSubtable DECIMAL(5,2),
        MinNumRowsPerDB INTEGER,
        AvgNumRowsPerDB INTEGER,
        MaxNumRowsPerDB INTEGER,
        LrgCyls BIGINT,
        SmlCyls BIGINT)
        MapName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL
PRIMARY INDEX ( TheDate ,TheTime ,TableID ,TableIDTAI ,DBSize ,
PctOfDBsInSubtable );
BTEQ -- Enter your SQL request or BTEQ command:

```

The next example shows the BTEQ output from running `PopulateFsysInfoTable` macro with the L display option, and empty strings for `target_database_name` and `target_table_name` to force the output to display on screen.

The column titles names of the output change when redirected to the screen instead of using the target table to store the output. For example, the column name is “PctDBsIn1to8” in the target table and the corresponding column is “% Of DBs In 1 to 8” when the output is redirected to the screen.

The warning that is returned after `PopulateFsysInfoTable` macro is expected behavior when the output is displayed to the screen.

---

#### Note:

The BTEQ `.foldline` and `.sidetitles` commands have been used to cause the column titles of the returned data set to be displayed on the side, and the corresponding values of those columns to be shown beside the column titles. This example shows seven rows of output. The first column of each row is the Date column.

```

exec populatefsysinfotable('Call_Logs', 'Daily_Log','showblocks','l','','');
*** Procedure has been executed.
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 27 seconds.

```

\*\*\* ResultSet# 1 : 7 rows returned by "SYSLIB.POPULATEFSYSINFOTABLESP".

```

      Date 13/12/22
      Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
      TID          0000E90A0000
TID//TAI          0
      Comp//Method MANUAL
      Comp//State N
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
      DBSize          2
      DBs Per//Size          4
% Of DBs In//SubTable 100.00
      Min Rows//Per DB          1
      Avg Rows//Per DB          1
      Max Rows//Per DB          1
      Lrg Cyls          4
      Sml Cyls
      Map//Name
      Date 13/12/22
      Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
      TID          0000E90A0000
TID//TAI          1024
      Comp//Method MANUAL
      Comp//State
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
      DBSize          0
      DBs Per//Size
% Of DBs In//SubTable
      Min Rows//Per DB
      Avg Rows//Per DB
      Max Rows//Per DB
      Lrg Cyls          4
      Sml Cyls
      Map//Name
      Date 13/12/22
      Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
TID          0000E90A0000

```



```

TID//TAI          1024
  Comp//Method MANUAL
  Comp//State U
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
      DBSize          96
      DBs Per//Size          1
% Of DBs In//SubTable  12.50
  Min Rows//Per DB          2221
  Avg Rows//Per DB          2221
  Max Rows//Per DB          2221
      Lrg Cyls
      Sml Cyls
      Map//Name
        Date 13/12/22
        Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
TID          0000E90A0000
TID//TAI          1024
  Comp//Method MANUAL
  Comp//State U
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
      DBSize          98
      DBs Per//Size          1
% Of DBs In//SubTable  12.50
  Min Rows//Per DB          2263
  Avg Rows//Per DB          2263
  Max Rows//Per DB          2263
      Lrg Cyls
      Sml Cyls
      Map//Name
        Date 13/12/22
        Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
TID          0000E90A0000
TID//TAI          1024
  Comp//Method MANUAL
  Comp//State U
Est Comp//Ratio(%)
Est Uncomp//DBS(%)
      DBSize          99

```

```

      DBs Per//Size              1
% Of DBs In//SubTable    12.50
      Min Rows//Per DB          2297
      Avg Rows//Per DB          2297
      Max Rows//Per DB          2297
      Lrg Cyls
      Sml Cyls
      Map//Name
      Date 13/12/22
      Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
      TID            0000E90A0000
      TID//TAI       1024
      Comp//Method MANUAL
      Comp//State U
      Est Comp//Ratio(%)
      Est Uncomp//DBS(%)
      DBSize          102
      DBs Per//Size              1
% Of DBs In//SubTable    12.50
      Min Rows//Per DB          2359
      Avg Rows//Per DB          2359
      Max Rows//Per DB          2359
      Lrg Cyls
      Sml Cyls
      Map//Name
      Date 13/12/22
      Time 21:46:49
      DB//Name Call_Logs
      Tbl//Name Daily_Log
      TID            0000E90A0000
      TID//TAI       1024
      Comp//Method MANUAL
      Comp//State U
      Est Comp//Ratio(%)
      Est Uncomp//DBS(%)
      DBSize          254
      DBs Per//Size              4
% Of DBs In//SubTable    50.00
      Min Rows//Per DB          5907
      Avg Rows//Per DB          5907
      Max Rows//Per DB          5907
      Lrg Cyls

```

```
Sml Cyls
Map//Name
```

## Example: Populate a File System Information Table for SHOWCOMPRESS Output, Option 'S'

Create table:

```
create multiset table Call_Logs.Daily_Log as dbc.tvdm with data;
```

Create data table that will contain SHOWCOMPRESS data:

```
exec
  createfsysinfotable('Call_Logs'
    , 'showcTrgTableShort'
    , 'perm'
    , 'y'
    , 'showcompress'
    , 'S'
    , 'TD_Map1');
```

To populate data into a target table using the stored procedure:

```
exec
  populatefsysinfotable('Call_Logs'
    , 'Daily_Log'
    , 'showcompress'
    , 's'
    , 'Call_Logs'
    , 'showcTrgTableShort');
```

```
*** Procedure has been executed.
```

To display data for the short display output using bteq with the .foldline and .sidetitles formatting qualifiers:

```
exec populatefsysinfotable('Call_Logs', 'Daily_Log', 'showcompress', 's', '', '');

*** Procedure has been executed.
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 1 second.

*** ResultSet# 1 : 4 rows returned by "SYSLIB.POPULATEFSYSINFOTABLESP".
```

```

        Date 17/12/08
        Time 07:16:07
        DB Name Call_Logs
        Tbl Name Daily_log
        TID 0000BF270000
        TID_TAI 1024
        TLA BLC Option DEFAULT
System BLC Option MANUAL
        TLA BLC ALG DEFAULT
        System BLC ALG ZLIB
        TLA BLC Level DEFAULT
System BLC Level 6
        Map Name TD_MAP1

```

```

        Date 17/12/08
        Time 07:16:07
        DB Name Call_Logs
        Tbl Name Daily_log
        TID 0000BF270000
        TID_TAI 2048
        TLA BLC Option DEFAULT
System BLC Option MANUAL
        TLA BLC ALG DEFAULT
        System BLC ALG ZLIB
        TLA BLC Level DEFAULT
System BLC Level 6
        Map Name TD_MAP1

```

```

        Date 17/12/08
        Time 07:16:07
        DB Name Call_Logs
        Tbl Name Daily_log
        TID 0000BF270000
        TID_TAI 2052
        TLA BLC Option DEFAULT
System BLC Option MANUAL
        TLA BLC ALG DEFAULT
        System BLC ALG ZLIB
        TLA BLC Level DEFAULT
System BLC Level 6
        Map Name TD_MAP1

```

```

        Date 17/12/08

```

```

Time 07:16:07
DB Name Call_Logs
Tbl Name Daily_log
TID 0000BF270000
TID_TAI 2056
TLA BLC Option DEFAULT
System BLC Option MANUAL
TLA BLC ALG DEFAULT
System BLC ALG ZLIB
TLA BLC Level DEFAULT
System BLC Level 6
Map Name TD_MAP1

```

## Example: Populate a File System Information Table for SHOWCOMPRESS Output, Option 'L'

Create table:

```
create multiset table Call_Logs.Daily_Log as dbc.tvn with data;
```

Create data table that will contain SHOWCOMPRESS data:

```

exec
  createfsysinfotable('Call_Logs'
    , 'showcTrgTableLong'
    , 'perm'
    , 'y'
    , 'showcompress'
    , 'l'
    , 'TD_Map1');

```

To populate data into a target table using the stored procedure:

```

exec
  populatefsysinfotable('Call_Logs'
    , 'Daily_Log'
    , 'showcompress'
    , 'l'
    , 'Call_Logs'
    , 'showcTrgTableLong');

```

\*\*\* Procedure has been executed.

To display data for the long display output using bteq with the .foldline and .sidetitles formatting qualifiers:

```
exec populatefsysinfotable('Call_Logs','Daily_Log','showcompress','1','','');
```

```
*** Procedure has been executed.
```

```
*** Warning: 3212 The stored procedure returned one or more result sets.
```

```
*** Total elapsed time was 2 seconds.
```

```
*** ResultSet# 1 : 9 rows returned by "SYSLIB.POPULATEFSYSINFOTABLESP".
```

```

      Date  17/12/08
      Time  07:11:01
      DB Name  Call_Logs
      Tbl Name  Daily_log
      TID  0000BF270000
      TID_TAI  -1
      TLA BLC Option  DEFAULT
      System BLC Option  MANUAL
      TLA BLC ALG  DEFAULT
      System BLC ALG  ZLIB
      TLA BLC Level  DEFAULT
      System BLC Level  6
      Map Name  TD_MAP1
      Comp State  COMPRESSED
      Comp Algo  IPPZLIB
      Comp Level  6
      Exact Comp Ratio(%)  85.38
      Exact (%) of Blocks  91.17
      Exact (%) of Data  88.28
      Exact Used GB

```

```

      Date  17/12/08
      Time  07:11:01
      DB Name  Call_Logs
      Tbl Name  Daily_log
      TID  0000BF270000
      TID_TAI  -1
      TLA BLC Option  DEFAULT
      System BLC Option  MANUAL
      TLA BLC ALG  DEFAULT
      System BLC ALG  ZLIB
      TLA BLC Level  DEFAULT
      System BLC Level  6
      Map Name  TD_MAP1

```

```

Comp State UNCOMPRESSED
Comp Algo N/A
Comp Level N/A
Exact Comp Ratio(%)
Exact (%) of Blocks      8.82
Exact (%) of Data       11.71
Exact Used GB

```

```

Date 17/12/08
Time 07:11:01
DB Name Call_Logs
Tbl Name Daily_log
TID 0000BF270000
TID_TAI 0
TLA BLC Option DEFAULT
System BLC Option MANUAL
TLA BLC ALG DEFAULT
System BLC ALG ZLIB
TLA BLC Level DEFAULT
System BLC Level 6
Map Name TD_MAP1
Comp State UNCOMPRESSED
Comp Algo N/A
Comp Level N/A
Exact Comp Ratio(%)
Exact (%) of Blocks      100.00
Exact (%) of Data       100.00
Exact Used GB

```

```

Date 17/12/08
Time 07:11:01
DB Name Call_Logs
Tbl Name Daily_log
TID 0000BF270000
TID_TAI 1024
TLA BLC Option DEFAULT
System BLC Option MANUAL
TLA BLC ALG DEFAULT
System BLC ALG ZLIB
TLA BLC Level DEFAULT
System BLC Level 6
Map Name TD_MAP1
Comp State COMPRESSED
Comp Algo IPPZLIB

```

```

Comp Level 6
Exact Comp Ratio(%) 87.31
Exact (%) of Blocks 100.00
  Exact (%) of Data 100.00
  Exact Used GB

Date 17/12/08
Time 07:11:01
DB Name Call_Logs
Tbl Name Daily_log
TID 0000BF270000
TID_TAI 1028
TLA BLC Option DEFAULT
System BLC Option MANUAL
TLA BLC ALG DEFAULT
System BLC ALG ZLIB
TLA BLC Level DEFAULT
System BLC Level 6
Map Name TD_MAP1
Comp State UNCOMPRESSED
Comp Algo N/A
Comp Level N/A
Exact Comp Ratio(%)
Exact (%) of Blocks 100.00
  Exact (%) of Data 100.00
  Exact Used GB

Date 17/12/08
Time 07:11:01
DB Name Call_Logs
Tbl Name Daily_log
TID 0000BF270000
TID_TAI 1032
TLA BLC Option DEFAULT
System BLC Option MANUAL
TLA BLC ALG DEFAULT
System BLC ALG ZLIB
TLA BLC Level DEFAULT
System BLC Level 6
Map Name TD_MAP1
Comp State UNCOMPRESSED
Comp Algo N/A
Comp Level N/A
Exact Comp Ratio(%)

```



```

Exact (%) of Blocks      100.00
  Exact (%) of Data      100.00
    Exact Used GB

      Date 17/12/08
      Time 07:11:01
      DB Name Call_Logs
      Tbl Name Daily_log
      TID 0000BF270000
      TID_TAI 2048
      TLA BLC Option DEFAULT
      System BLC Option MANUAL
      TLA BLC ALG DEFAULT
      System BLC ALG ZLIB
      TLA BLC Level DEFAULT
      System BLC Level 6
      Map Name TD_MAP1
      Comp State COMPRESSED
      Comp Algo IPPZLIB
      Comp Level 6
Exact Comp Ratio(%)      87.31
Exact (%) of Blocks      100.00
  Exact (%) of Data      100.00
    Exact Used GB

      Date 17/12/08
      Time 07:11:01
      DB Name Call_Logs
      Tbl Name Daily_log
      TID 0000BF270000
      TID_TAI 2052
      TLA BLC Option DEFAULT
      System BLC Option MANUAL
      TLA BLC ALG DEFAULT
      System BLC ALG ZLIB
      TLA BLC Level DEFAULT
      System BLC Level 6
      Map Name TD_MAP1
      Comp State COMPRESSED
      Comp Algo IPPZLIB
      Comp Level 6
Exact Comp Ratio(%)      54.00
Exact (%) of Blocks      100.00
  Exact (%) of Data      100.00

```

```

Exact Used GB

      Date 17/12/08
      Time 07:11:01
      DB Name Call_Logs
      Tbl Name Daily_log
      TID 0000BF270000
      TID_TAI 2056
      TLA BLC Option DEFAULT
System BLC Option MANUAL
      TLA BLC ALG DEFAULT
System BLC ALG ZLIB
      TLA BLC Level DEFAULT
System BLC Level 6
      Map Name TD_MAP1
      Comp State COMPRESSED
      Comp Algo IPPZLIB
      Comp Level 6
Exact Comp Ratio(%) 61.00
Exact (%) of Blocks 100.00
Exact (%) of Data 100.00
Exact Used GB

```

## Example: Populate a File System Information Table for SHOWWHERE Output, Option 'S'

To populate data into a target table for the short display output:

```

exec
dbc.populatefsysinfotable('Call_Logs','t','showwhere','s','Call_Logs','showw_s')
;
sel * from Call_Logs.showw_s;

      TheDate 15/01/28
              TheTime 01:00:57.780000
      DataBaseName Call_Logs
      TableName t
      TableID 0000130B0000
      TableIDTAI 0
              Vproc -1
              Drive -1
              LrgCyls 10
              SmlCyls 0

```

```

      MapName
TableType PERM
      Grade
      GradePct .00
      VHCyls   0
      HotCyls  0
      WarmCyls 0
      ColdCyls 10
      CylsinSSD 0
TemperaturesMature Y

```

To populate data onto the screen for short display:

```

exec dbc.populatefsysinfotable('Call_Logs','Daily_Log','showwhere','s','','');
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 6 seconds

```

```

*** ResultSet# 1 : 1 rows returned by "SYSLIB.POPULATEFSYSINFOTABLESP".

```

```

      Date 15/01/28
      Time 09:01:46
      DB Name Call_Logs
Tbl Name Daily_Log
      TID 0000130B0000
      TID_TAI 0
      Vproc -1
      Drive -1
      Lrg Cyls 10
      Sml Cyls
      MapName
TableType PERM
      Grade
      Grade(%)
      VH Cyls
      Hot Cyls
      Warm Cyls
      Cold Cyls 10
      Cyls in SSD
Temperatures Mature Y

```

## Example: Populate a File System Information Table for SHOWWHERE Output, Option 'M'

To populate data into a target table for the medium display output:

```
exec dbc.populatefsysinfotable('Call_Logs',
'Daily_Log','showwhere','m','Call_Logs','showw_m');
```

```
SELECT * from Call_Logs.showw_m;
```

```

      TheDate  15/01/28
                TheTime  01:03:12.160000
                DataBaseName  Call_Logs
      TableName  Daily_Log
                TableID  0000130B0000
      TableIDTAI      0
                Vproc    0
                Drive    -1
                LrgCyls   2
                SmlCyls   0
                MapName
      TableType  PERM
                Grade
      GradePct      .00
                VHCyls    0
                HotCyls   0
                WarmCyls  0
                ColdCyls  2
                CylsinSSD  0
TemperaturesMature  Y

                TheDate  15/01/28
                TheTime  01:03:12.160000
      DataBaseName  Call_Logs
      TableName    Daily_Log
      TableID      0000130B0000
      TableIDTAI    0
                Vproc    2
                Drive    -1
                LrgCyls   3
                SmlCyls   0
                MapName
      TableType    PERM
```

```

                                Grade
                                GradePct      .00
                                VHCyls        0
                                HotCyls       0
                                WarmCyls      0
                                ColdCyls      3
                                CylsinSSD     0
TemperaturesMature Y
                                TheDate      15/01/28
                                TheTime      01:03:12.160000
                                DataBaseName  Call_Logs
                                TableName     Daily_Log
                                TableID      0000130B0000
                                TableIDTAI   0
                                Vproc        1
                                Drive        -1
                                LrgCyls      2
                                SmlCyls      0
                                MapName
                                TableType    PERM
                                Grade
                                GradePct      .00
                                VHCyls        0
                                HotCyls       0
                                WarmCyls      0
                                ColdCyls      2
                                CylsinSSD     0
TemperaturesMature Y
                                TheDate      15/01/28
                                TheTime      01:03:12.160000
                                DataBaseName  Call_Logs
                                TableName     Daily_Log
                                TableID      0000130B0000
                                TableIDTAI   0
                                Vproc        3
                                Drive        -1
                                LrgCyls      3
                                SmlCyls      0
                                MapName
                                TableType    PERM
                                Grade
                                GradePct      .00
                                VHCyls        0
                                HotCyls       0

```

```

WarmCyls      0
ColdCyls      3
CylsinSSD     0
TemperaturesMature Y

```

To populate data onto the screen for Medium display:

```

exec dbc.populatefsysinfotable('Call_Logs','Daily_Log','showwhere','m','','');
*** Warning: 3212 The stored procedure returned one or more result sets.

```

```

Date 15/01/28
Time 09:04:14
DB Name Call_Logs
Tbl Name Daily_Log
TID 0000130B0000
TID_TAI 0
Vproc 0
Drive -1
Lrg Cyls 2
Sml Cyls
MapName
TableType PERM
Grade
Grade(%)
VH Cyls
Hot Cyls
Warm Cyls
Cold Cyls 2
Cyls in SSD
Temperatures Mature Y
Date 15/01/28
Time 09:04:14
DB Name Call_Logs
Tbl Name Daily_Log
TID 0000130B0000
TID_TAI 0
Vproc 1
Drive -1
Lrg Cyls 2
Sml Cyls
MapName
TableType PERM
Grade
Grade(%)
VH Cyls
Hot Cyls
Warm Cyls
Cold Cyls 2
Cyls in SSD
Temperatures Mature Y
Date 15/01/28
Time 09:04:14
DB Name Call_Logs
Tbl Name Daily_Log
TID 0000130B0000
TID_TAI 0
Vproc 2
Drive -1
Lrg Cyls 3
Sml Cyls
MapName
TableType PERM
Grade

```

```

      Grade(%)
      VH Cyls
      Hot Cyls
      Warm Cyls
      Cold Cyls      3
      Cyls in SSD
Temperatures Mature Y
      Date 15/01/28
      Time 09:04:14
      DB Name Call_Logs
      Tbl Name Daily_Log
      TID 0000130B0000
      TID_TAI 0
      Vproc 3
      Drive -1
      Lrg Cyls 3
      Sml Cyls
      MapName
      TableType PERM
      Grade
      Grade(%)
      VH Cyls
      Hot Cyls
      Warm Cyls
      Cold Cyls      3
      Cyls in SSD
Temperatures Mature Y

```

## Example: Populate a File System Information Table for SHOWWHERE Output, Option 'L'

To populate data into a target table for the long display output:

```

exec dbc.populatefsysinfotable('Call_Logs',
'Daily_Log','showwhere','l','Call_Logs','showw_l');
sel * from Call_Logs.showw_l;

      TheDate 15/01/28
      TheTime 01:06:52.710000
      DataBaseName Call_Logs
      TableName Daily_Log
      TableID 0000130B0000
      TableIDTAI 0
      Vproc 2
      Drive 2
      LrgCyls 3
      SmlCyls 0
      MapName
      TableType PERM
      Grade
      GradePct .00
      VHCyls 0

```

```

        HotCyls      0
        WarmCyls     0
        ColdCyls     3
        CylsinSSD    0
TemperaturesMature Y
        TheDate      15/01/28
        TheTime      01:06:52.710000
        DataBaseName Call_Logs
        TableName    Daily_Log
        TableID       0000130B0000
        TableIDTAI    0
                        Vproc 1
                        Drive 1
        LrgCyls       2
        SmlCyls       0
        MapName
        TableType     PERM
                        Grade
        GradePct      .00
                        VHCyls 0
        HotCyls       0
        WarmCyls      0
        ColdCyls      2
        CylsinSSD     0
TemperaturesMature Y
        TheDate      15/01/28
        TheTime      01:06:52.710000
        DataBaseName Call_Logs
        TableName    Daily_Log
        TableID       0000130B0000
        TableIDTAI    0
                        Vproc 0
                        Drive 0
        LrgCyls       2
        SmlCyls       0
        MapName
        TableType     PERM
                        Grade
        GradePct      .00
                        VHCyls 0
        HotCyls       0
        WarmCyls      0
        ColdCyls      2
        CylsinSSD     0

```



```

TemperaturesMature Y
      TheDate 15/01/28
      TheTime 01:06:52.710000
      DataBaseName Call_Logs
      TableName Daily_Log
      TableID 0000130B0000
      TableIDTAI 0
      Vproc 3
      Drive 3
      LrgCyls 3
      SmlCyls 0
      MapName
      TableType PERM
      Grade
      GradePct .00
      VHCyls 0
      HotCyls 0
      WarmCyls 0
      ColdCyls 3
      CylsinSSD 0
TemperaturesMature Y

```

To populate data onto the screen for Long display:

```

exec dbc.populatefsysinfotable('Call_Logs', 'Daily_Log','showwhere','l','','');
*** Warning: 3212 The stored procedure returned one or more result sets.

```

```

Date 15/01/28
      Time 09:07:54
      DB Name Call_Logs
      Tbl Name Daily_Log
      TID 0000130B0000
      TID_TAI 0
      Vproc 0
      Drive 0
      Lrg Cyls 2
      Sml Cyls
      MapName
      TableType PERM
      Grade
      Grade(%)
      VH Cyls
      Hot Cyls
      Warm Cyls
      Cold Cyls 2
      Cyls in SSD
Temperatures Mature Y
      Date 15/01/28
      Time 09:07:54
      DB Name Call_Logs
      Tbl Name Daily_Log
      TID 0000130B0000
      TID_TAI 0

```

```

Vproc 1
Drive 1
Lrg Cyls 2
Sml Cyls
MapName
TableType PERM
Grade
Grade(%)
VH Cyls
Hot Cyls
Warm Cyls
Cold Cyls 2
Cyls in SSD
Temperatures Mature Y
Date 15/01/28
Time 09:07:54
DB Name Call_Logs
Tbl Name Daily_Log
TID 0000130B0000
TID_TAI 0
Vproc 2
Drive 2
Lrg Cyls 3
Sml Cyls
MapName
TableType PERM
Grade
Grade(%)
VH Cyls
Hot Cyls
Warm Cyls
Cold Cyls 3
Cyls in SSD
Temperatures Mature Y
Date 15/01/28
Time 09:07:54
DB Name Call_Logs
Tbl Name Daily_Log
TID 0000130B0000
TID_TAI 0
Vproc 3
Drive 3
Lrg Cyls 3
Sml Cyls
MapName
TableType PERM
Grade
Grade(%)
VH Cyls
Hot Cyls
Warm Cyls
Cold Cyls 3
Cyls in SSD
Temperatures Mature Y

```

## Examples Using CLASSSPOOL and SHOWWHERE

A LOCAL line is listed with class SPOOL and class SPOOLPOOL when the the PopulateFSysInfoTable macro is run with the CLASSSPOOL and SHOWWHERE options.

## Example: Populate a File System Information Table for SHOWWHERE and SPOOL CLASS Output, 'S' Option

Create target table and populate data into the table for short display output. Several columns on the right side of the output are not included because of space constraint.

```
exec dbc.createfsysinfotable('db','spools','perm','y','showwhere','s','td_map1');
exec dbc.populatefsysinfotable('dbc','classspool','showwhere','s','db','spools');
sel * from db.spools;
```

Date	Time	DB Name	Tbl Name	TID	TID_TAI	Vproc	Drive	Lrg	Cyls	Sml	Cyls	TableType	Grade	Grade(%)
21/01/13	05:17:34.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	688	0	0	0	SPOOL	LOCAL	100.00
21/01/13	05:17:34.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOL	SLOW	.00
21/01/13	05:17:34.000000	DBC	CLASS SPOOLPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOLPOOL	MEDIUM	.00
21/01/13	05:17:34.000000	DBC	CLASS SPOOLPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOLPOOL	SLOW	.00
21/01/13	05:17:34.000000	DBC	CLASS SPOOLPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOLPOOL	FAST	.00
21/01/13	05:17:34.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOL	MEDIUM	.00
21/01/13	05:17:34.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOL	FAST	.00

Display data for short display with output to the screen. Several columns on the right side of the output are not included because of space constraint.

```
exec dbc.populatefsysinfotable('dbc','classspool','showwhere','s','','');
```

Date	Time	DB Name	Tbl Name	TID	TID_TAI	Vproc	Drive	Lrg	Cyls	Sml	Cyls	TableType	Grade	Grade(%)
21/01/13	02:03:44.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	684	0	0	0	SPOOL	FAST	100.00
21/01/13	02:03:44.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOL	LOCAL	.00
21/01/13	02:03:44.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOL	MEDIUM	.00
21/01/13	02:03:44.000000	DBC	CLASS SPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOL	SLOW	.00
21/01/13	02:03:44.000000	DBC	CLASS SPOOLPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOLPOOL	FAST	.00
21/01/13	02:03:44.000000	DBC	CLASS SPOOLPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOLPOOL	MEDIUM	.00
21/01/13	02:03:44.000000	DBC	CLASS SPOOLPOOL	000000000000	0	-1	-1	0	0	0	0	SPOOLPOOL	SLOW	.00

## Example: Populate a File System Information Table for SHOWWHERE and SPOOL CLASS Output, 'M' Option

Create target table and populate data into the table for medium display output. Several columns on the right side of the output are not included because of space constraint.

```
exec dbc.createfsysinfotable('db','spools','perm','y','showwhere','m','td_map1');
exec dbc.populatefsysinfotable('dbc','classspool','showwhere','m','db','spoolm');
sel * from db.spoolm;
```

Date	Time	DB Name	Tbl Name	TID	TID_TAI	Vproc	Drive	Lrg	Cyls	Sml	Cyls	TableType	Grade	Grade(%)
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	0	-1	0	0	0	0	SPOOL	SLOW	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	0	-1	342	0	0	0	SPOOL	LOCAL	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	-1	0	0	0	0	SPOOLPOOL	MEDIUM	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	1	-1	342	0	0	0	SPOOL	LOCAL	100.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	-1	0	0	0	0	SPOOLPOOL	SLOW	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	1	-1	0	0	0	0	SPOOL	FAST	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	1	-1	0	0	0	0	SPOOL	SLOW	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	-1	0	0	0	0	SPOOLPOOL	FAST	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	-1	0	0	0	0	SPOOLPOOL	MEDIUM	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	-1	0	0	0	0	SPOOLPOOL	SLOW	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	1	-1	0	0	0	0	SPOOL	MEDIUM	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	-1	0	0	0	0	SPOOLPOOL	FAST	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	1	-1	0	0	0	0	SPOOL	MEDIUM	.00
21/01/13	05:20:19.000000	DBC	CLASS SPOOL	000000000000	0	1	-1	0	0	0	0	SPOOL	FAST	.00

Display data for medium display with output to the screen. Several columns on the right side of the output are not included because of space constraint.

```
exec dbc.populatefsysinfotable('dbc','classspool','showwhere','m','','');
```

Date	Time	DB Name	Tbl Name	TID	TID_TAI	Vproc	Drive	Lrg	Cyls	Sml	Cyls	TableType	Grade	Grade(%)
21/01/13	02:04:44.000000	DBC	CLASS SPOOL	000000000000	0	0	-1	342	0	0	0	SPOOL	FAST	100.00
21/01/13	02:04:44.000000	DBC	CLASS SPOOL	000000000000	0	0	-1	0	0	0	0	SPOOL	LOCAL	.00

21/01/13	02:04.44.000000	DBC	CLASS	SPOOL	000000000000	0	0	-1							SPOOL	MEDIUM	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOL	000000000000	0	0	-1							SPOOL	SLOW	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOL	000000000000	0	1	-1							SPOOL	FAST	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOL	000000000000	0	1	-1	342						SPOOL	LOCAL	100.00
21/01/13	02:04.44.000000	DBC	CLASS	SPOOL	000000000000	0	1	-1							SPOOL	MEDIUM	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOL	000000000000	0	1	-1							SPOOL	SLOW	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOLPOOL	000000000000	0	1	-1							SPOOLPOOL	FAST	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOLPOOL	000000000000	0	1	-1							SPOOLPOOL	MEDIUM	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOLPOOL	000000000000	0	1	-1							SPOOL	SLOW	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOLPOOL	000000000000	0	1	-1							SPOOLPOOL	FAST	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOLPOOL	000000000000	0	1	-1							SPOOLPOOL	MEDIUM	
21/01/13	02:04.44.000000	DBC	CLASS	SPOOLPOOL	000000000000	0	1	-1							SPOOLPOOL	SLOW	

## Example: Populate a File System Information Table for SHOWWHERE and SPOOL CLASS Output, 'L' Option

Create target table and populate data into the table for large display output. Several columns on the right side of the output are not included because of space constraint.

```
exec dbc.createfsysinfotable('dbc','spools','perm','y','showwhere','l','td_map1');
exec dbc.populatefsysinfotable('dbc','classspool','showwhere','l','db','spool1');
sel * from db.spool1;
```

Date	Time	DB Name	Tbl Name	TID	TID_TAI	Vproc	Drive	Lrg	Cyls	Sml	Cyls	TableType	Grade	Grade(%)
21/01/13	05:21.36.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	5	0	0	0	0	SPOOLPOOL	SLOW	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	5	0	0	0	0	SPOOLPOOL	FAST	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	1	5	0	0	0	0	SPOOL	MEDIUM	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	1	5	0	0	0	0	SPOOL	SLOW	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	5	0	0	0	0	SPOOLPOOL	MEDIUM	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	1	5	342	0	0	0	SPOOL	LOCAL	100.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	1	5	0	0	0	0	SPOOL	FAST	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	2	0	0	0	0	SPOOLPOOL	SLOW	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	2	0	0	0	0	SPOOLPOOL	MEDIUM	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	2	0	0	0	0	SPOOLPOOL	FAST	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	0	2	342	0	0	0	SPOOL	LOCAL	100.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	0	2	0	0	0	0	SPOOL	SLOW	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	0	2	0	0	0	0	SPOOL	MEDIUM	.00
21/01/13	05:21.36.000000	DBC	CLASS SPOOL	000000000000	0	0	2	0	0	0	0	SPOOL	FAST	.00

Display data for long display with output to the screen. Several columns on the right side of the output are not included because of space constraint.

```
exec dbc.populatefsysinfotable('dbc','classspool','showwhere','l','','');
sel * from db.spool1;
```

Date	Time	DB Name	Tbl Name	TID	TID_TAI	Vproc	Drive	Lrg	Cyls	Sml	Cyls	TableType	Grade	Grade(%)
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	0	2					SPOOL	FAST	
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	0	2	342				SPOOL	LOCAL	100.00
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	0	2					SPOOL	MEDIUM	
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	0	2					SPOOL	SLOW	
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	1	5					SPOOL	FAST	
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	1	5	342				SPOOL	LOCAL	100.00
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	1	5					SPOOL	MEDIUM	
21/01/13	02:04.57.000000	DBC	CLASS SPOOL	000000000000	0	1	5					SPOOL	SLOW	
21/01/13	02:04.57.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	2					SPOOLPOOL	FAST	
21/01/13	02:04.57.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	2					SPOOLPOOL	MEDIUM	
21/01/13	02:04.57.000000	DBC	CLASS SPOOLPOOL	000000000000	0	0	2					SPOOLPOOL	SLOW	
21/01/13	02:04.57.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	5					SPOOLPOOL	FAST	
21/01/13	02:04.57.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	5					SPOOLPOOL	MEDIUM	
21/01/13	02:04.57.000000	DBC	CLASS SPOOLPOOL	000000000000	0	1	5					SPOOLPOOL	SLOW	

## AlterFsysInfoTable\_TD16/AlterFsysInfoTable\_ANSI\_TD16

The AlterFsysInfoTable\_TD16 macro alters existing FsysInfo tables created in Teradata Database Releases 15.00 or 15.10.

The SQL SHOW\* functions display the column MapName. AlterFsysInfoTable\_TD16 converts the existing target tables created in Teradata Database 15.0 or 15.10 to include columns in the SHOWBLOCKS/SHOWWHERE SQL output. The macro creates an additional column, MapName, in the SQL SHOW\* target tables.

On a Teradata Database 16.00 system, use the `PopulateFsysInfoTable_TD15` macro to populate the pre-existing target table or alter the table to 16.00, and then populate the table using the new macros.

---

**Note:**

If you use ANSI session mode, use the `_ANSI` form of the macro.

---

## Required Privileges

In addition to the privileges mentioned above, to run this macro you must have EXECUTE privileges on the `AlterFsysInfoTable_TD16` macro or on the database containing the macro.

The macro name and arguments are case-insensitive.

## Syntax

```
[DBC.] { AlterFsysInfoTable | AlterFsysInfoTable_ANSI } (
  'target_database_name',
  'target_table_name',
  { 'SHOWBLOCKS' | 'SHOWWHERE' },
  'display_opt'
)
```

## Syntax Elements

### *target\_database\_name*

The database in which the previous version of the target table exists.

---

**Note:**

You must have appropriate privileges to create tables in the target database.

---

### *target\_table\_name*

The name of the target table that must be altered to have the "MapName" column in it.

---

**Note:**

You must have appropriate privileges to create tables in the target database.

---

## SHOWBLOCKS

## SHOWWHERE

The listed SHOW\* command includes a MapName column. The MapName column appears in the listed SHOW\* command displays, and is populated with map name values.

***display\_opt***

The level of file system detail that can be stored in the table.

- S provides minimal amount of file system information
- M provides a medium amount of file system information
- L provides the maximum amount of file system information

For more information about the specific information displayed, see [PopulateFsysInfoTable/PopulateFsysInfoTable\\_ANSI](#).

**Note:**

The display option chosen for CreateFsysInfoTable must match the display option to be used for the corresponding execution of the PopulateFsysInfoTable macro.

If output is directed to the screen (for example, empty strings are specified for database and table names), then table creation is not required using CreateFsysInfoTable or CreateFsysInfoTable\_ANSI macro call. Any display option can be used.

## Examples

### Example: Using the PopulateFsysInfoTable Macro to Populate a Pre-Existing Table

On a Teradata Database Release 16.00 system, you can use the PopulateFsysInfoTable\_TD15 macro to populate a pre-existing table, or alter tables to Teradata Database 16.00, and then populate using the updated PopulateFsysInfoTable.

An example for the successful output on the SHOWBLOCKS target table:

```
SHOW TABLE SystemInfo.showb_s;  <-- This table is an existing table created in
Teradata Database 15.0/15.10.

CREATE SET TABLE SystemInfo.showb_s ,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO,
  MAP = TD_MAP1
(
  TheDate DATE FORMAT 'YY/MM/DD',
  TheTime TIME(6),
  DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
  TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
```

```

TableID BYTE(6),
TableIDTAI INTEGER,
CompressionMethod CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
CompressionState VARCHAR(2) CHARACTER SET LATIN NOT CASESPECIFIC,
EstCompRatio DECIMAL(5,2),
EstPctOfUncompDBs DECIMAL(5,2),
PctDBsIn1to8 BYTEINT,
PctDBsIn9to24 BYTEINT,
PctDBsIn25to64 BYTEINT,
PctDBsIn65to120 BYTEINT,
PctDBsIn121to168 BYTEINT,
PctDBsIn169to216 BYTEINT,
PctDBsIn217to256 BYTEINT,
PctDBsIn257to360 BYTEINT,
PctDBsIn361to456 BYTEINT,
PctDBsIn457to512 BYTEINT,
PctDBsIn513to760 BYTEINT,
PctDBsIn761to1024 BYTEINT,
PctDBsIn1025to1034 BYTEINT,
PctDBsIn1035to1632 BYTEINT,
PctDBsIn1633to2048 BYTEINT,
MinDBSize INTEGER,
AvgDBSize INTEGER,
MaxDBSize INTEGER,
TotalNumDBs BIGINT,
LrgCyls BIGINT,
SmlCyls BIGINT)
PRIMARY INDEX ( TheDate ,TheTime ,TableID ,CompressionState );

```

BTEQ -- Enter your SQL request or BTEQ command:

```
exec alterfssysteminfotable_td16('SystemInfo','showb_s','showblocks','s');
```

```
exec alterfssysteminfotable_td16('SystemInfo','showb_s','showblocks','s');  <--
Macro successfully completed.
```

BTEQ -- Enter your SQL request or BTEQ command:

```
show table SystemInfo.showb_s;  <-- This same table has the column 'MapName'
at the end of the original table definition.
```

```
CREATE SET TABLE SystemInfo.showb_s ,FALLBACK ,
NO BEFORE JOURNAL,
```

```

NO AFTER JOURNAL,
CHECKSUM = DEFAULT,
DEFAULT MERGEBLOCKRATIO,
MAP = TD_MAP1
(
  TheDate DATE FORMAT 'YY/MM/DD',
  TheTime TIME(6),
  DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
  TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
  TableID BYTE(6),
  TableIDTAI INTEGER,
  CompressionMethod CHAR(8) CHARACTER SET LATIN NOT CASESPECIFIC,
  CompressionState VARCHAR(2) CHARACTER SET LATIN NOT CASESPECIFIC,
  EstCompRatio DECIMAL(5,2),
  EstPctOfUncompDBs DECIMAL(5,2),
  PctDBsIn1to8 BYTEINT,
  PctDBsIn9to24 BYTEINT,
  PctDBsIn25to64 BYTEINT,
  PctDBsIn65to120 BYTEINT,
  PctDBsIn121to168 BYTEINT,
  PctDBsIn169to216 BYTEINT,
  PctDBsIn217to256 BYTEINT,
  PctDBsIn257to360 BYTEINT,
  PctDBsIn361to456 BYTEINT,
  PctDBsIn457to512 BYTEINT,
  PctDBsIn513to760 BYTEINT,
  PctDBsIn761to1024 BYTEINT,
  PctDBsIn1025to1034 BYTEINT,
  PctDBsIn1035to1632 BYTEINT,
  PctDBsIn1633to2048 BYTEINT,
  MinDBSize INTEGER,
  AvgDBSize INTEGER,
  MaxDBSize INTEGER,
  TotalNumDBs BIGINT,
  LrgCyls BIGINT,
  SmlCyls BIGINT,
  MapName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC)  <--MapName
column was created.
PRIMARY INDEX ( TheDate ,TheTime ,TableID ,CompressionState );

```

An example for the successful output on the SHOWWHERE target table:

```
SHOW TABLE SystemInfo.showw_m;
```



```

SHOW TABLE SystemInfo.showw_m;

CREATE SET TABLE SystemInfo.showw_m ,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    TheDate DATE FORMAT 'YY/MM/DD',
    TheTime TIME(6),
    DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TableID BYTE(6),
    TableIDTAI SMALLINT,
    Vproc SMALLINT,
    Drive INTEGER,
    LrgCyls BIGINT,
    SmlCyls BIGINT,
    TableType VARCHAR(12) CHARACTER SET UNICODE UPPERCASE NOT CASESPECIFIC
NOT NULL,
    Grade VARCHAR(6) CHARACTER SET UNICODE UPPERCASE NOT CASESPECIFIC
NOT NULL,
    GradePct DECIMAL(5,2),
    VHCyls BIGINT,
    HotCyls BIGINT,
    WarmCyls BIGINT,
    ColdCyls BIGINT,
    CylsinSSD BIGINT,
    TemperaturesMature VARCHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX ( TheDate ,TheTime ,TableID ,TableIDTAI ,LrgCyls ,
SmlCyls );

BTEQ -- Enter your SQL request or BTEQ command:
exec alterfssysteminfotable_td16('SystemInfo','showw_m','showwhere','m');

exec alterfssysteminfotable_td16('SystemInfo','showw_m','showwhere','m');

*** Procedure has been executed.
*** Total elapsed time was 5 seconds.

BTEQ -- Enter your SQL request or BTEQ command:

```

```

show table SystemInfo.showw_m;

show table SystemInfo.showw_m;

*** Text of DDL statement returned.
*** Total elapsed time was 1 second.

CREATE SET TABLE SystemInfo.showw_m ,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO,
    MAP = TD_MAP1
(
    TheDate DATE FORMAT 'YY/MM/DD',
    TheTime TIME(6),
    DataBaseName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TableName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC NOT NULL,
    TableID BYTE(6),
    TableIDTAI SMALLINT,
    Vproc SMALLINT,
    Drive INTEGER,
    LrgCyls BIGINT,
    SmlCyls BIGINT,
    TableType VARCHAR(12) CHARACTER SET UNICODE UPPERCASE NOT CASESPECIFIC
NOT NULL,
    Grade VARCHAR(6) CHARACTER SET UNICODE UPPERCASE NOT CASESPECIFIC
NOT NULL,
    GradePct DECIMAL(5,2),
    VHCyls BIGINT,
    HotCyls BIGINT,
    WarmCyls BIGINT,
    ColdCyls BIGINT,
    CylsinSSD BIGINT,
    TemperaturesMature VARCHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
    MapName VARCHAR(128) CHARACTER SET UNICODE NOT CASESPECIFIC)
PRIMARY INDEX ( TheDate ,TheTime ,TableID ,TableIDTAI ,LrgCyls ,
SmlCyls );

```

If the new populated macro is used without altering the table, the following error displays:

```

BTEQ -- Enter your SQL request or BTEQ command:
exec dbc.PopulateFsysInfoTable('CallData', 'Weekly', 'showblocks', 's',

```

```
'SystemInfo', 'SHOWB_Short' );

exec dbc.PopulateFsysInfoTable('CallData', 'Weekly', 'showblocks', 's',
'SystemInfo', 'SHOWB_Short' );
*** Failure 3813 POPULATEFSYSINFOTABLESP:The positional assignment list has too
many values.
*** Total elapsed time was 4 seconds.
```

## Example: Migrating Tables to a Prior Release

If the table created in Teradata Database Release 16.00 with the updated `PopulateFsysInfoTable` macro is migrated to Release 15.00 or 15.10, is it not possible to populate the table using the old macros.

The following error displays:

```
BTEQ -- Enter your SQL request or BTEQ command:
exec dbc.PopulateFsysInfoTable('CallData', 'Weekly', 'showblocks', 's',
'SystemInfo', 'SHOWB_S' );

exec dbc.PopulateFsysInfoTable('CallData', 'Weekly', 'showblocks', 's',
'SystemInfo', 'SHOWB_S' );
*** Failure 3812 POPULATEFSYSINFOTABLESP:The positional assignment list has too
few values.
*** Total elapsed time was 5 seconds.
```

You can still query the table to get historical `SHOWBLOCKS` or `SHOWWHERE` data. However, the new tables must be created and then populated, or you can alter the existing tables manually to drop the "MapName" column before populating `SHOWBLOCKS` or `SHOWWHERE` data. The alter table statement is as follows:

```
ALTER TABLE SystemInfo.SHOWB_S drop MapName;
```

## Heatmap Table Function and Macro (tdheatmap and tdheatmap\_m)

The `tdheatmap` table function provides a frequency of access report for database objects on a specific AMP. To use this function, the Temperature-Based Block-Level Compression feature must be enabled, or you must have licensed the Teradata Virtual Storage feature. With `tdheatmap`, you can view the relative temperature of tables and data, the tables or cylinders targeted for the Teradata Intelligent Memory (TIM) cache, and the temperature ranges the tables occupy.

Data temperature reflects the frequency of data access. Hot data is the most frequently accessed. Cold data is the least frequently accessed. Warm data is moderately accessed.

There are different ways to invoke tdheatmap:

- `SELECT * FROM TABLE (tdheatmap(amp_number)) AS t1`

You also can select particular columns when using the tdheatmap function, but not when using the tdheatmap\_m macro.

- Use the tdheatmap\_m macro.
- Use the tdheatmap function to create a view to invoke the heatmap on the same AMP for subsequent invocations.

Use the view to SELECT particular columns or to use the WHERE clause.

## Syntax

```
tdheatmap ( amp_number )
```

## Syntax Elements

### *amp\_number*

The AMP number on which tdheatmap is to be executed.

## Differences Between the tdheatmap Function, Macro and View

There are differences between invoking the tdheatmap function, macro, or creating a view. The following table describes the differences to help you choose which method to use.

	Table function tdheatmap	Macro tdheatmap_ m	View tdheatmap_v
Ensures that the heatmap report is always taken from the same AMP.			X
Ability to check heatmap report on different amps if any skew is suspected.	X	X	
Ability to select only particular columns to display.	X		X
Use of 'StartTableIdUniq' to join to DBC.TVM.TVMID so as to get English name of the objects. This is required because the operator does not output English description of database and table names, and you must write an additional query to obtain this.	X	X	X
Ability to specify the names of database objects to limit the scope of the heatmap report to generate.	X		X

## Returned Columns

The following columns can be obtained from the `tdheatmap` function.

### General Fields

*TheDate*

The date the heatmap was produced.

*TheTime*

The time the heatmap was produced.

*AmpNumber*

The VPROC identifier of the AMP on which the heatmap was originally produced.

*DatabaseName*

The name of the database.

*TableName*

The name of the table.

*CylinderId(Lsw+Msw)*

The unique identifier of cylinder.

### Starting Table

*StartTableId*

*StartTableIdUniq* and *StartTableIdTypeAndIndex*.

*StartTableIdUniq*

The unique ID that identifies the starting table. It can be joined to `dbc.tvm.tvmid`.

*StartTableIdTypeAndIndex*

The subtable type of the starting table: Primary, Fallback, Secondary. For example, 1024 is the Primary data, 1028 is the Secondary Index, 2048 is the Primary fallback.

*StartPartition*

Internal partition number of the starting row.

---

#### Note:

This is not an external partition.

---

*StartRowId*

The rowId(*hash0* + *hash1* + *uniq0* + *uniq1*) of the starting row.

**Ending Table***EndTableId**EndTableIdUniq* and *EndTableTypeAndIndex*.*EndTableIdUniq*

The unique ID that identifies the ending table. It can be joined to *dbc.tvm.tvmid*.

*EndTableTypeAndIndex*

Subtable type of the ending table: Primary, Fallback, Secondary. For example, 1024 is the Primary data, 1028 is the Secondary Index, 2048 is the Primary fallback.

*EndPartition*

Internal partition number of the ending row.

**Note:**

This is not the external partition associated with PARTITION keyword via SQL.

*EndRowHash*

The row hash value (*hash0* and *hash1*) of the ending row.

**Temperature Information***Temperature*

The TVS temperature in the float point.

*NormalizedTempInfo*

If Temperature is greater than *warmceiling*, it is classified as 'HOT'. Any Temperature between *warmceiling* and *warmfloor* is 'WARM,' and the rest is COLD.

*RequestedTempInfo*

The requested temperature by Vantage when File System requests the allocation of a cylinder.

*TempPercentile*

Percentage value indicates how many percent permanent cylinders are colder than this cylinder.

*TempWarmCeiling*

The upper limit of the warm/hot temperature boundary.

*TempWarmFloor*

The lower limit of the warm/hot temperature boundary.

*TempVeryHotFloor*

The lowest temperature above which temperature can be considered to belong to *VeryHotCache*.

*PercentFull*

Percentage value indicates how much percent of the extent is full.

**TIM Cache***VeryHotCandidate*

Yes or No. TVS identifies whether the selected cylinder is the hot candidate. If TIM is not activated, this column displays BLANK.

*VeryHotCache*

Yes or No. The cylinder is (or is not) actually in *VeryHotCache*. If TIM is not activated, this column displays BLANK.

**Storage Placement***Grade*

Describes whether the cylinder storage media is SLOW, MEDIUM, or FAST. If TVS is not enabled on the system, this column displays BLANK.

*MediaType*

Specifies the cylinder storage media type as DISK, FILE, SSD, NETWORK, or OTHER. Each value indicates whether the data is on a rotating disk drive, a flat file, a solid state drive, a network-attached storage device, or another media type.

*StorageClass*

Specifies the cylinder storage media class as PRIMARY or SECONDARY. The default is PRIMARY.

**Usage Notes**

The heatmap function requires either Teradata Virtual Storage (TVS) or Temperature Based Block Level Compression (TBBLC) enabled on the system.

**Examples**

## Example: Using the tdheatmap Function

You can use the tdheatmap function to return just a few important columns, like the table name, cylinder ID, and temperature.

```
SELECT date TheDate,
       t1.ampnumber,
       dbase.databasenamei DatabaseName,
       tvn.tvnnamei TableName,
       to_byte(t1.cylinderidmsw) || to_byte(t1.cylinderidlsb) CylinderId,
       t1.temperature
FROM TABLE(tdheatmap(2)) AS t1,
     dbc.tvn tvn,
     dbc.dbase dbase
WHERE t1.starttableiduniq = tvn.tvnid
AND   tvn.databaseid = dbase.databaseid;
```

TheDate	AmpNumber	DatabaseName	TableName	CylinderId	Temperature
11/10/2016	2	TD_SYSFNLIB	BITNOT	00-06-00-02-00-00-00-05	10.58
11/10/2016	2	DW123	ALPHA	00-06-00-02-00-00-00-46	10
11/10/2016	2	TD_SYSFNLIB	WEEKNUMBER_ OF_ CALENDAR_TSWZ	00-06-00-02-00-00-00-45	17.83
11/10/2016	2	DW123	ALPHA	00-06-00-02-00-00-00-49	10
11/10/2016	2	DBC	CHANGEDROWJOURNAL	00-06-00-02-00-00-00-21	10.06
11/10/2016	2	DW123	ALPHA	00-06-00-02-00-00-00-4A	10
11/10/2016	2	DBC	ALL	00-06-00-02-00-00-00-23	10
11/10/2016	2	DW123	ALPHA	00-06-00-02-00-00-00-47	10
11/10/2016	2	DBC	TVFIELDS	00-06-00-02-00-00-00-3A	10.21



#### 4: File System Information Macros and Functions

TheDate	AmpNumber	DatabaseName	TableName	CylinderId	Temperature
11/10/2016	2	DW123	MILLIONS	00-06-00-02-00-00-00-4B	10
11/10/2016	2	DW123	ALPHA	00-06-00-02-00-00-00-48	10
11/10/2016	2	DW123	ALPHA	00-06-00-02-00-00-00-4E	10

## Example: Using the tdheatmap\_m Macro

The tdheatmap\_m macro is another way to get the temperature information. Run the macro to display additional columns such as Date, Time, DatabaseName, TableName, and CylinderId.

Note that when using the macro, you cannot SELECT particular columns or use the WHERE clause.

The macro tdheatmap\_m is created automatically using the tdheatmap table function. It takes the AMP number as an input parameter and is created in the DBC database.

#### 4: File System Information Macros and Functions

The following example uses the `tdheatmap_m` macro to generate a heatmap for AMP 1:

```
EXECUTE DBC.tdheatmap_m(1);
```

And the resulting output:

Comprehensive System Performance Metrics Report - Q3 2024																														
The Date	The Time	Amp #	Database Name	Table Name	Start Table Id	Start Table IdUqn	Start Table IdType And Index	Start Partition	Start RowId	End Table	End Table IdUqn	End Table IdType And Index	End Partition	End Row Hash	CylinderId	Cylinder IdLsw	Cylinder IdHsw	Temp	Normalized TempIn/o	Requested TempIn/o	Ver/Hot Candidate	Ver/Hot Cache	Temp Warm Ceiling	Temp Warm Floor	Temp Ver/Hot Floor	Percent Full	Temp Percentile	Grade	Media Type	Storage Class
2/14/2016	23:00:19	1	DVDRESCRI	RESCRIE10	00-00-0A-D6-04-00	00-00-D6-0A	1,024		01-C6-C5-6D-00-00-00-01	00-00-0A-D6-04-00	00-00-D6-0A	1,024		01-F1-21-92	00-06-00-01-00-00-0F-21	393,217	3,873	0.02	COLD	WARM	N	N	2,419.81	375.85	4,313.83	90.2	0	FAST	DISK	PRIMARY
2/14/2016	23:00:19	1	TD_SYSFNLI	GREATEST2	00-00-06-00-00-00	00-00-06-06	0		00-00-00-00-00-00-00-01	00-00-0A-09-08-00	00-00-09-0A	2,048		8C-8B-0 F1-39	00-06-00-01-00-00-02-C8	393,217	712	43.23	COLD	HOT	N	N	2,419.81	375.85	4,313.83	16.74	0	MEDIUM	DISK	PRIMARY
2/14/2016	23:00:19	1	SYSLIB	LDL_CLEAN	00-00-0A-0F-00-00	00-00-0F-0A	0		01-00-00-00-00-00-00-01	00-00-0A-D6-04-00	00-00-D6-0A	1,024		00-0F-04E-9B	00-06-00-01-00-00-09-10	393,217	2,320	0.16	COLD	WARM	N	N	2,419.81	375.85	4,313.83	88.21	0	MEDIUM	DISK	PRIMARY
2/14/2016	23:00:19	1	DVDRESCRI	RESCRIE10	00-00-0A-D6-04-00	00-00-D6-0A	1,024		02-37-A5-92-00-00-00-01	00-00-0A-D6-04-00	00-00-D6-0A	1,024		02-C2-0 C8-5C	00-06-00-01-00-00-09-1E	393,217	2,334	0.02	COLD	WARM	N	N	2,419.81	375.85	4,313.83	90.13	0	SLOW	DISK	PRIMARY
2/14/2016	23:00:19	1	DBC	MIGRATION	70-00-00-00-00-00	70-00-00-00	0		00-00-00-00-00-00-00-01	00-00-01-85-04-00	00-00-85-01	1,024		FB-6E-0 C0-91	00-06-00-01-00-00-00-21	393,217	33	6,110.40	VERY HOT	VERY HOT	Y	Y	2,419.81	375.85	4,313.83	62.41	98	FAST	DISK	PRIMARY
2/14/2016	23:00:19	1	DVDRESCRI	RESCRIE10	00-00-0A-D6-04-00	00-00-D6-0A	1,024		03-41-88-29-00-00-00-01	00-00-0A-D6-04-00	00-00-D6-0A	1,024		03-8B-02-F8	00-06-00-01-00-00-09-22	393,217	2,338	0.02	COLD	WARM	N	N	2,419.81	375.85	4,313.83	90.2	0	SLOW	DISK	PRIMARY
2/14/2016	23:00:19	1	DBC	UDFINFO	00-00-01-65-04-00	00-00-65-01	1,028		00-24-A8-AB-00-00-00-00-01	00-00-06-05-00-00	00-00-05-06	0		00-00-00-00-01	00-06-00-01-00-00-00-D7	393,217	215	48.17	COLD	HOT	N	N	2,419.81	375.85	4,313.83	5.27	0	MEDIUM	DISK	PRIMARY
2/14/2016	23:00:19	1	DVDRESCRI	RESCRIE10	00-00-0A-D6-04-00	00-00-D6-0A	1,024		02-1A-0A-C8-00-00-00-00-01	00-00-0A-D6-04-00	00-00-D6-0A	1,024		02-41-0 CE-09	00-06-00-01-00-00-09-1B	393,217	2,331	0.02	COLD	WARM	N	N	2,419.81	375.85	4,313.83	84.74	0	SLOW	DISK	PRIMARY
2/14/2016	23:00:19	1	DBC	ALL	00-00-00-00-00-00	00-00-00-00	0		00-00-00-00-00-00-00-01	00-00-00-6F-08-00	00-00-6F-00	2,048		BD-5D-0 B1-BA	00-06-00-01-00-00-00-23	393,217	35	6,089.51	VERY HOT	VERY HOT	Y	Y	2,419.81	375.85	4,313.83	12.55	98	FAST	DISK	PRIMARY
2/14/2016	23:00:19	1	DVDRESCRI	RESCRIE10	00-00-0A-D6-04-00	00-00-D6-0A	1,024		02-22-C8-89-00-00-00-00-01	00-00-0A-D6-04-00	00-00-D6-0A	1,024		02-ED-0 A4-4C	00-06-00-01-00-00-09-1F	393,217	2,335	0.03	COLD	WARM	N	N	2,419.81	375.85	4,313.83	90.2	0	SLOW	DISK	PRIMARY

## Example: Using tdheatmap to Create a View

Use the `tdheatmap` function to construct a view to periodically query and get snapshots of AMP-level data temperature changes over time. This allows you to see trends in the data access patterns.

The definition of the view below includes all of the columns returned by the heatmap table function, with the addition of the Date, Time, Database name, Table name and Cylinder ID fields. Replace *target database name* with the name of the designated view database, and replace *amp\_number* with the number of the AMP the function needs to run on.

```
REPLACE view target database name.tdheatmap_v as
SELECT  date TheDate
        , time TheTime
        , t1.ampnumber
        , dbase.databasenamei DatabaseName
        , tvn.tvnnamei TableName
        , t1.starttableid
        , t1.starttableiduniq
        , t1.starttableidtypeandindex
        , t1.startpartition
        , t1.startrowid
        , t1.endtableid
        , t1.endtableiduniq
        , t1.endtableidtypeandindex
        , t1.endpartition
        , t1.endrowhash
        , to_byte(t1.cylinderidmsw) || to_byte(t1.cylinderidls)w) CylinderId
        , t1.cylinderidmsw
        , t1.cylinderidls
        , t1.temperature
        , t1.normalizedtempinfo
        , t1.requestedtempinfo
        , t1.veryhotcandidate
        , t1.veryhotcache
        , t1.tempwarmceiling
        , t1.tempwarmfloor
        , t1.tempveryhotfloor
        , t1.percentfull
        , t1.temppercentile
        , t1.grade
        , t1.mediatype
        , t1.storageclass from table (syslib.tdheatmap (amp_number))
as t1 (
```

```

        AmpNumber,
        StartTableId,
        StartTableIdUniq,
        StartTableIdTypeAndIndex,
        StartPartition,
        StartRowId,
        EndTableId,
        EndTableIdUniq,
        EndTableIdTypeAndIndex,
        EndPartition,
        EndRowHash,
        CylinderIdMsw,
        CylinderIdLsw,
        Temperature,
        NormalizedTempInfo,
        RequestedTempInfo,
        VeryHotCandidate,
        VeryHotCache,
        TempWarmCeiling,
        TempWarmFloor,
        TempVeryHotFloor,
        PercentFull,
        TempPercentile,
        Grade,
        MediaType,
        StorageClass),
dbc.tvn tvn,
dbc.dbase dbase
where t1.starttableiduniq = tvn.tvnid
      and tvn.databaseid = dbase.databaseid
;

GRANT SELECT on dbc.tvn to target database name with GRANT option;
GRANT SELECT on dbc.dbase to target database name with GRANT option;
GRANT EXECUTE function on syslib.tdheatmap to target database name with
GRANT option;

```

Using a view is useful for tracking data temperature trends over time. You can use the view to create a heatmap history table:

```

/* CREATE the history table with the first set of heatmap */
CREATE TABLE target database name.heatmap_history
AS

```

```
(SELECT * FROM target database name.tdheatmap_v)
WITH DATA
PRIMARY INDEX (TheDate, TheTime)
```

```
/* Load the subsequent heatmap data */
INSERT INTO target database name.heatmap_history
SELECT * from target database name.tdheatmap_v;
```

## Example: Data Temperature Report Showing a Specific Table

The following is an example of a heatmap report showing the temperature property of cylinder ID '0006000100000038'.

### Note:

Only one cylinder is selected for ease of illustration.

```
SELECT * FROM demo.tdheatmap_v WHERE DatabaseName = 'EMPLOYEE' AND TableName =
'BASIC_INFO' ORDER BY Temperature DESC
```

Field	Results
TheDate	9/23/2015
TheTime	20:46:49
AmpNumber	1
DatabaseName	EMPLOYEE
TableName	BASIC_INFO
StartTableId	00-00-0A-A8-04-0A
StartTableIdUniq	00-00-A8-0A
StartTableIdTypeAndIndex	1,024
StartPartition	0
StartRowId	DB-BA-AF-F3-00-00-00-01
EndTableId	00-00-0A-A8-04-0A
EndTableIdUniq	00-00-A8-0A
EndTableIdTypeAndIndex	1,024
EndPartition	0

Field	Results
EndRowHash	F0-30-4E-94
CylinderId	00-06-00-01-00-00-00-38
CylinderIdMsw	393,217
CylinderIdLsw	56
Temperature	29.41
NormalizedTempInfo	WARM
RequestedTempInfo	WARM
VeryHotCandidate	
VeryHotCache	
TempWarmCeiling	113.42
TempWarmFloor	18.11
TempVeryHotFloor	0
PercentFull	90.72
TempPercent	47.00
Grade	MEDIUM
MediaType	DISK
StorageClass	PRIMARY

This sample heatmap report indicates the following:

1. The cylinder belongs to a primary permanent table:

```
StartTableIdTypeAndIndex = 1024
```

2. TIM is not enabled:

```
VeryHotCandidate = BLANK
```

and

```
VeryHotCache = BLANK
```

and

```
TempVeryHotFloor = 0
```

3. This cylinder is not a mixed cylinder:

```
StartTableIdUniq = 0000A80A
```

and

```
EndTableIdUniq = 0000A80A
```

If it is a mixed cylinder, StartTableIdUniq and EndTableIdUniq shows different values.

4. This heatmap was taken from AMP 1:

```
AmpNumber = 1
```

5. This table is not partitioned:

```
StartPartition = 0
```

and

```
EndPartition = 0
```

## Example: Showing Cylinders in TIM Cache

The following example shows the cylinders of the BASIC\_INFO table that are in TIM cache.

Database Name	Table Name	StartTable IdType AndIndex	Temp	Normalized TempInfo	Very Hot Candidate	Very Hot Cache	% Full
DBC	ALL	0	120.99	HOT	Y	Y	31.6
TD_	TO_TIMESTAMP	0	102.38	WARM	Y	Y	81.28
SYSFNLIB							
DBC	UDFINFO	0	101.29	WARM	Y	Y	11.98
DBC	UDFINFO_TD14	0	91.42	WARM	Y	Y	52.93
EMPLOYEE	BASIC_INFO	1,024	86.27	WARM	Y	Y	90.72
EMPLOYEE	BASIC_INFO	1,024	29.41	WARM	Y	Y	90.72
EMPLOYEE	BASIC_INFO	1,024	18.67	WARM	Y	Y	90.14
EMPLOYEE	BASIC_INFO	1,024	18.11	WARM	Y	Y	90.24
EMPLOYEE	BASIC_INFO	1,024	18.11	COLD	Y	Y	90.1
EMPLOYEE	BASIC_INFO	1,024	18.06	COLD	Y	Y	90.64
EMPLOYEE	BASIC_INFO	1,024	18	COLD	Y	Y	90.17
EMPLOYEE	BASIC_INFO	1,024	17.75	COLD	Y	N	90.48
EMPLOYEE	BASIC_INFO	1,024	17.73	COLD	Y	N	90.32
EMPLOYEE	BASIC_INFO	1,024	17.73	COLD	Y	N	90.62
EMPLOYEE	BASIC_INFO	1,024	17.72	COLD	Y	Y	90.26



Database Name	Table Name	StartTable IdType AndIndex	Temp	Normalized TempInfo	Very Hot Candidate	Very Hot Cache	% Full
EMPLOYEE	BASIC_INFO	1,024	17.72	COLD	Y	Y	90.18

## Example: Heatmap Report Showing Tables with Different Subtable Types

The following example displays a heatmap report showing tables with different subtable types. The report shows the Primary FALLBACK subtable (2048) and the SECONDARY INDEX subtable (1028) of 'EMPLOYEE.BASIC\_INFO,' along with its Primary permanent tables.

The Date	The Time	Amp No	Database Name	Table Name	Start Table Id Uniq	Start Table Id Type And Index	End Table Id Uniq	End Table Id Type And Index*	Cyl Id	Temp	Normalized Temp Info
9/24/2015	0:38:19	2	DBC	ALL	00-00-00-00	0	00-00-38-01	0	00-06-00-02-00-00-00-20	266.23	HOT
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-55	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-56	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-61	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-63	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-6B	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-58	252.42	WARM

#### 4: File System Information Macros and Functions

The Date	The Time	Amp No	Database Name	Table Name	Start Table Id Uniq	Start Table Id Type And Index	End Table Id Uniq	End Table Id Type And Index*	Cyl Id	Temp	Normalized Temp Info
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-57	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-60	252.42	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-65	252.41	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-5A	252.41	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-59	17252.41.75	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-62	252.41	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-64	252.41	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-SE-0D	1,024	00-06-00-02-00-00-00-53	252.36	WARM

#### 4: File System Information Macros and Functions

The Date	The Time	Amp No	Database Name	Table Name	Start Table Id Uniq	Start Table Id Type And Index	End Table Id Uniq	End Table Id Type And Index*	Cyl Id	Temp	Normalized Temp Info
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-6F	252.34	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	2,048	00-00-A8-0A	<b>2,048</b>	00-06-00-02-00-00-00-6D	252.34	WARM
9/24/2015	0:38:19	2	DBC	DBQLRULETBL	00-00-4C-01	0	00-00-54-07	0	00-06-00-02-00-00-00-4A	226.45	WARM
9/24/2015	0:38:19	2	DBC	TVM	00-00-38-01	1,024	00-00-48-01	2,052	00-06-00-02-00-00-00-1F	218.82	WARM
9/24/2015	0:38:19	2	TD_SYSFNLIB	ARRAY_MUL_B	00-00-55-07	0	00-00-A8-0A	1,024	00-06-00-02-00-00-00-48	73.75	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	<b>1,028</b>	00-06-00-02-00-00-00-37	37.09	WARM
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-40	35.72	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-4C	35.55	COLD

#### 4: File System Information Macros and Functions

The Date	The Time	Amp No	Database Name	Table Name	Start Table Id Uniq	Start Table Id Type And Index	End Table Id Uniq	End Table Id Type And Index*	Cyl Id	Temp	Normalized Temp Info
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-3C	20.88	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-41	20.58	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-4B	20.46	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-54	20.02	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-3B	19.97	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-49	19.96	COLD
9/24/2015	0:38:19	2	EMPLOYEE	BASIC_INFO	00-00-A8-0A	1,024	00-00-A8-0A	1,024	00-06-00-02-00-00-00-3F	19.95	COLD

## Related Information

- [CreateFsysInfoTable/CreateFsysInfoTable\\_ANSI](#) for information about SHOWWHERE.
- *Teradata Vantage™ - Database Utilities*, B035-1102 for details about the DIP script.
- *Teradata Vantage™ - Database Administration*, B035-1093.

# Map Functions, Macros, and Procedures

You can use maps to reduce downtime when adding or changing the AMPs in a configuration, or to improve performance when using small tables.

With contiguous or sparse maps, you can define which tables remain in the current configuration, and which tables move to maps in to a new configuration. You can also redistribute tables for the new maps over a period of time, instead of all at the same time.

Contiguous and sparse maps are also used to run table operators on a subset of AMPs. The SYSLIB database contains functions, and the TDMAPS database contain procedures, that provide information about contiguous and sparse maps.

## Map Functions

Maps provide the following functions:

- SYSLIB.ContiguousMapAMPs
- SYSLIB.SparseMapAMPs
- SYSLIB.SparseTableAMPs

## SYSLIB.ContiguousMapAMPs

SYSLIB.ContiguousMapAMPs is a system-provided table function. It returns the primary and fallback AMPs in a contiguous map.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[SYSLIB.] ContiguousMapAMPs ('MapNameString')
```

### Syntax Elements

**SYSLIB.**

Name of the database where the function is located.

***MapNameString***

The name of the contiguous map.

## Usage Notes

The *MapNameString* is VARCHAR(128) whose value is a map name.

### Example: Identifying Which AMPs are in a Contiguous Map

Use the function SYSLIB.ContiguousMapAMPs to view the AMPs in a contiguous map.

In this example, the first column is the AMP number. The second column contains a two-character code indicating whether the AMP is the primary ('P'), fallback ('F'), or both primary and fallback ('PF') AMP.

```
SELECT * FROM TABLE (ContiguousMapAmps('TD_Map1')) dt;
```

AmpNo	AmpType
-----	-----
0	PF
1	PF
2	PF
3	PF
4	PF
5	PF
6	PF
7	PF

## SYSLIB.SparseMapAMPs

SYSLIB.SparseMapAMPs is a table function that returns the AMPs with a sparse map defined for a table, join index, or hash index.

The view DBC.SparseMapAMPsV is an alternative to using the table function SYSLIB.SparseMapAMPs. The underlying system view, DBC.ObjectsInSparseMapsV, prepares all arguments for the table function. See *Teradata Vantage™ - Data Dictionary*, B035-1092 for more details about views DBC.SparseMapAMPsV and DBC.ObjectsInSparseMapsV.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
[SYSLIB.] SparseMapAMPs (
  'DatabaseName',
  'ObjectName',
  'ObjectKind',
```



```
' MapSlot' ,
' NumberOfPrimaryAmps' ,
' ColocationName'
)
```

## Syntax Elements

### **SYSLIB.**

Name of the database where the function is located.

### ***DatabaseName***

The database in which the object is defined.

### ***ObjectName***

Name of the table, join index, or hash index defined in *DatabaseName*. It cannot be NULL.

### ***ObjectKind***

The kind of objects, such as a table, join index or hash index, defined in the sparse map.

### ***MapSlot***

The map slot of the sparse map.

If the map containing the map slot does not exist, or is not a sparse map, an error is returned.

### ***NumberOfPrimaryAMPs***

The number of primary AMPs defined in the sparse map.

### ***ColocationName***

The colocation name string for the table.

## Usage Notes

The SYSLIB.SparseMapAMPs table function results contain five columns and one row per AMP. The columns include the:

- Database name
- Object name
- Object kind
- AMP number
- Two-character code indicating whether the AMP is primary ('P'), fallback ('F'), or both primary and fallback ('PF')

## Example: Identifying an AMP with a Specific Table

Use the SYSLIB.SparseMapAMPs function to find which AMP contains a specific table in a sparse map.

```
WITH dt1 (DatabaseName, ObjectName, ObjectKind, MapSlot, NumberOfPrimaryAMPs,
ColocationName) AS
(SELECT DatabaseName, TableName, TableKind, MapSlot, NumberOfAMPs,
        ColocationName
 FROM DBC.TablesV t, DBC.Maps m
 WHERE ColocationName IS NOT NULL
        AND t.MapName = m.MapName)
SELECT AmpNo, PF, CAST(DbName AS CHAR(5)), CAST(ObjName AS CHAR(5))
FROM TABLE (SparseMapAMPs(dt1.DatabaseName, dt1.ObjectName, dt1.ObjectKind,
dt1.MapSlot, dt1.NumberOfPrimaryAMPs, dt1.ColocationName)) dt
ORDER BY 1;
```

AmpNo	PF	DbName	ObjName
0	P	db1	tabs2
7	P	db1	tabs1
24	F	db1	tabs2
31	F	db1	tabs1
72	F	db1	tabs2
96	P	db1	tabs2

```
WITH dt1 (DatabaseName, ObjectName, ObjectKind, MapSlot, NumberOfPrimaryAMPs,
ColocationName) AS
(SELECT DatabaseName, TableName, TableKind, MapSlot, NumberOfAMPs,
        ColocationName
 FROM DBC.TablesV t, DBC.Maps m
 WHERE ColocationName IS NOT NULL
        AND t.MapName = m.MapName)
SELECT CAST(DbName AS CHAR(12)), CAST(ObjName AS CHAR(12)), AmpNo, PF
FROM TABLE (SparseMapAMPs(dt1.DatabaseName, dt1.ObjectName,
dt1.ObjectKind, dt1.MapSlot, dt1.NumberOfPrimaryAMPs,
dt1.ColocationName)) dt
ORDER BY 1,2;
```

DbName	ObjName	AmpNo	PF
db1	tabs1	7	P
db1	tabs1	31	F

db1	tabs2	0	P
db1	tabs2	24	F
db1	tabs2	72	F
db1	tabs2	96	P

## SYSLIB.SparseTableAMPs

SYSLIB.SparseTableAMPs is a function that returns the AMPs on which a table, join index, or hash index with a sparse map is defined.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
[SYSLIB.] SparseTableAMPs (
  'DatabaseNameString',
  'ObjectNameString'
)
```

### Syntax Elements

#### SYSLIB.

Name of the database where the function is located.

#### DatabaseNameString

The name of the database. The data type is VARCHAR(128).

#### ObjectNameString

The name of a table, join index, or hash index. The data type is VARCHAR(128).

## Usage Notes

Like SHOW statements, privilege on the object is required.

The result is a VARCHAR(64000) CHARACTER SET LATIN string containing a fixed seven-character right-justified character representation for each AMP (primary and fallback), up to a maximum of 1024 AMPs.

Errors are returned if:

- The object does not exist.
- The passed-in object is defined in a contiguous map.

- The function is part of a query that references a table. The function can only be used in a query such as:

```
SELECT SYSLIB.SparseTableAMPs('db1','tab1') dt;
```

## Example: Identifying AMPs for a Table with a Sparse Map

Use the function SYSLIB.SparseTableAMPs to identify the AMPs for a specific table in a sparse map.

```
SELECT SYSLIB.SparseTableAMPs('db1','tab1') AS AMPList;
```

AMPList

```
-----
      1      202      543      32454
```

Use the following query to convert the function's result to a comma separated form.

```
SELECT REGEXP_REPLACE(TRIM(LEADING FROM
                          SYSLIB.SparseTableAMPs('db1','tab1')),' +', ',') AS AMPList;
```

AMPList

```
-----
1,202,543,32454
```

## Map Macros

The following macros are used to add or remove databases to an exclude list.

The macros are created when the DIP script is executed and reside in the TDMaps database.

Macro Name	Purpose	Data Type
AddDBInExcludeList	Add a database to the exclude list in SpaceExcludeDBListTbl. User can execute this macro for databases that do not need space re-adjustments by using the AdjustSpace procedure.	VARCHAR(128) CHARACTER SET UNICODE
DelDBInExcludeList	Remove a database from the exclude list in SpaceExcludeDBListTbl. User can execute this macro for removing a database name from the exclude list so that the AdjustSpace procedure can re-adjust the space settings.	VARCHAR(128) CHARACTER SET UNICODE

## Map Procedures

The TDMaps database stores metadata for moving tables on to new maps. It contains procedures that perform automation tasks related to system expansion.

In addition to the AdjustSpace procedure, the database uses two other types, Advisor and Mover procedures. Advisor procedures analyze user tables within logged query plans and make recommendations for moving sets of tables onto maps. The output recommendations can be customized and input to the Mover, which moves data for a list of tables onto new maps.

## AdjustSpace

After moving tables to maps or expanding a system, the database space limits may no longer be accurate or reflect the actual space usages. Use the AdjustSpace procedure to re-adjust the space and skew values for a user or database.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] AdjustSpace (
    'DatabaseName',
    'BufferPercent',
    'ZeroSkewForSubMap',
    'OutInfo'
)
```

### Syntax Elements

#### *DatabaseName*

The name of the database with the maximum permanent space and permanent skew to adjust based on its current usage.

If 'NULL,' all databases in the system are considered for adjustment, except for the databases listed in TDMaps.SpaceExcludeDBListTbl.

#### *BufferPercent*

he percentage of extra buffer space to retain in the database.

If NULL, the default values is fetched from TDMaps.SpaSettingsTbl.

**ZeroSkewForSubmap**

Sets zero skew if all the tables are in same map, and that map is a subset of TD\_GLOBALMAP. Valid values are *true*, *false*, *t*, *f*, and *NULL*.

**OutInfo**

Returns success or failure information. Additional output is logged into TDDMaps.SpaceLogTbl.

**Argument Types**

Expressions for *BufferPercent* must have the data type VARCHAR.

**Result Type**

The result row type for *OutInfo* is VARCHAR.

**Usage Notes**

The AdjustSpace procedure re-adjusts both the maximum permanent space and skew limit values of one or more databases. The re-adjustment is based on the database's current space usage and data distribution across AMPs.

Use the AdjustSpace procedure for users or databases after a system expansion, when the maximum space limits are increased. After analyzing all of the underlying tables within the user or database for map usage and altering them as recommended, use AdjustSpace to adjust the space and skew settings.

The permanent space quota is modified to be as near as possible to its current permanent space usage leaving some extra buffer space. The new permanent space quota is adjusted to a nearest multiple of default space chunk size.

After executing the procedure, the space columns (perm quota, current perm, allocated perm and perm skew) of the database before and after modification is recorded in TDDMaps.SpaceLogTable.

Databases with 0 as the current space are not modified by this procedure.

An error is reported if the DBS control flag LegacySpaceAcctg is set to TRUE, because setting a skew to non-zero value is not permitted with that setting.

**Examples****Example: Adjusting the Space Settings After a System Expansion**

The example database uses the following space values:

Amp No	Allocated Perm Space	Quota Perm Space	Skew limit
0	800	1000	10
1	900	1000	10
2	1100	1000	10
3	1000	1000	10

The system is then expanded, adding two new AMPs. Each AMP has 1000 units of space and uses a 100% prorated distribution during the reconfiguration:

Amp No	Allocated Perm Space	Quota Perm Space	Skew limit
0	800	1000	10
1	900	1000	10
2	1100	1000	10
3	1000	1000	10
4	0	1000	10
5	0	1000	10

After executing the AdjustSpace procedure for the database, the space values are changed:

Amp No	Allocated Perm Space	Quota Perm Space	Skew limit
0	800	700	75
1	900	700	75
2	1100	700	75
3	1000	700	75
4	0	700	75
5	0	700	75

### Example: Using AdjustSpace

To adjust space settings of all users and databases in the system (except DBC) with default settings, use:

```
CALL TDMaps.AdjustSpace(NULL, NULL, NULL, outlog);
```

To adjust space settings of a specific user or database with default settings, use:

```
CALL TDMaps.AdjustSpace('SALES_DB', NULL, NULL, outlog);
```

To adjust space settings of a specific user or database with a 10% buffer space percentage, use:

```
CALL TDMaps.AdjustSpace('SALES_DB', 10, NULL, outlog);
```

### Using the ZeroSkewForSubMap Parameter

You can use the ZeroSkewForSubMap parameter when reconfiguring space for users in a system. For example, if you have a system with 100 AMPs that have 100 users in the system, you can reconfigure to add 100 more AMPs. You can then add the extra space to the users.

After reconfiguring, give the users the extra space by passing NULL as the DatabaseName parameter, which automatically releases the extra space and set based on how the data is skewed.

```
CALL TDMaps.AdjustSpace(NULL, NULL, 'true', out1);
```

## Advisor Procedures

The Advisor procedures analyze tables and recommend which tables to move to specific maps. The procedures are helpful for moving large- and medium-sized tables from one contiguous map to another. The procedures are also helpful for moving small tables to a sparse map.

Advisor decides whether to move sets of tables to the destination map as a group, analyzing the table size, and makes recommendations on moving small tables to sparse maps.

### AddExclusionListEntrySP(X)

Adds an entry to the Exclusion List of databases, tables, join indexes or hash indexes.

This procedure works in conjunction with CreateExclusionListSP. CreateExclusionListSP is called once to create an empty list, and then AddExclusionListEntrySP(X) is called repeatedly to add objects to that list.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] AddExclusionListEntrySP(X) (
  'ExclusionListName',
  'DatabaseName',
  { 'ObjectName' | NULL },
  :NumObjectsAdded
) [;]
```



## Syntax Elements

### **TDMaps.**

The name of the database.

### ***ExclusionListName***

Name of the exclusion list.

### ***DatabaseName***

The database in which the object is defined.

### ***ObjectName***

Name of the table, join index, or hash index defined in *DatabaseName*. *ObjectName* can be NULL, in which case all tables, join indexes, and hash indexes defined in *DatabaseName* are included in the list.

### ***NumObjectsAdded***

This output parameter is an INTEGER representing the number of exclusions added to the list.

## Argument Type

Expressions passed to this procedure must have the following data types:

- *ExclusionListName*=VARCHAR
- *DatabaseName*=VARCHAR
- *ObjectName*=VARCHAR

## Usage Notes

A call to AddExclusionListEntry returns the total number of Exclusions successfully added to the list. ExclusionListName must be specified.

Caller-specified values for input parameters DatabaseName and ObjectName may contain wildcard characters (\_, %). Wildcard characters are resolved dynamically at the time the list is passed as a parameter to a called procedure. Wildcards are evaluated using standard SQL LIKE predicate semantics.

Defined object lists are stored in TDMaps tables ExclusionListTbl and ExclusionListEntry. If AddExclusionListEntrySPX is called, then defined objects you can access are stored in TDMaps tables ExclusionListTbl and ExclusionListEntry.

No automatic updates are made to existing objects lists stored in TDMaps as the result of subsequent DDL statements that DROP objects.

## Examples

### *Example: Creating an Exclusion List Named 'MyExclusions'*

In this example, you create an exclusion list named 'MyExclusions'. 'MyExclusions' consists of all tables in the 'Payroll' database, individual table 'Employee.Withholding', and all tables in database 'Taxes' whose names begin with 'Rate'.

```
CALL TDMaps.CreateExclusionListSP('MyExclusions', NULL, :ExclusionListId);
CALL TDMaps.AddExclusionListEntrySP('MyExclusions',
  'Payroll', NULL, :NumObjectsAdded);
CALL TDMaps.AddExclusionListEntrySP('MyExclusions',
  'Employee', 'Withholding', :NumObjectsAdded);
CALL TDMaps.AddExclusionListEntrySP('MyExclusions',
  'Taxes', 'Rate%', :NumObjectsAdded);
```

### *Example: Analyzing Exclusions Defined in 'MyExclusions'*

Use this example to analyze exclusions defined in 'MyExclusions' within all logged queries, and to make recommendations for moving them to map 'MyExclusionsMap'.

```
CALL TDMaps.AnalyzeSP(
  'TDMList1', 'MyObjects', 'MyExclusions',
  NULL, NULL, NULL, 'MyObjectssActions',
  :NumAlters, :NumExcludes, :NumLogEntries);
```

Procedure has been executed.

\*\*\* Total elapsed time was 20 minutes and 10 seconds.

NumAlters	NumExcludes	NumLogEntries
-----	-----	-----
10	15	3054

## AddMapListEntrySP(X)

Adds maps to a map list. The list of maps is a parameter to AnalyzeSP.

This procedure works in conjunction with CreateMapListSP. CreateMapListSP is called once to create an empty list, and then AddMapListEntrySP(X) is called repeatedly to add individual maps to that list.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] AddMapListEntrySP(X) (
  'MapListName',
  'MapName'
) [;]
```

## Syntax Elements

### TDMaps.

The name of the database.

### MapListName

Name of the map list.

### MapName

Name of the map to add.

## Argument Types

Expressions passed to this procedure must have the following data types:

- *MapListName*=VARCHAR
- *MapName*=VARCHAR

## Usage Notes

The MapListName and MapName must be specified.

Defined map lists are stored in TDMaps tables MapListTbl and MapListEntry. If AddMapListEntrySPX is called, then defined maps you can access are stored in TDMaps tables MapListTbl and MapListEntry.

No automatic updates are made to existing map lists stored in TDMaps as the result of subsequent DDL statements that drop maps.

A list of maps can only have one contiguous map.

All of the sparse maps in the list must have a different number of AMPs.

**Example: Calling AddMapListEntrySP(X)**

To call the procedure:

```
CALL tdmmaps.AddMapListEntrySP('MyMapList', 'MySparseMap');
```

**AddObjectListEntrySP(X)**

Adds individual objects to a list.

This procedure works in conjunction with CreateObjectListSP. CreateObjectListSP is called once to create an empty list, then AddObjectListEntrySP is called repeatedly to add individual objects to that list.

**ANSI Compliance**

This statement is a Teradata extension to the ANSI SQL:2011 standard.

**Syntax**

```
CALL [TDMaps.] AddObjectListEntrySP(X) (
  'ObjectName',
  'DatabaseName',
  { 'ObjectName' | NULL },
  :NumObjectsAdded
) [;]
```

**Syntax Elements****TDMaps.**

The name of the database.

**ObjectName**

Name of the object list. It must be specified.

**DatabaseName**

The database in which the object is defined.

**ObjectName**

Name of the table, join index, or hash index defined in *DatabaseName*. *ObjectName* can be NULL, in which case all tables, join indexes, and hash indexes defined in *DatabaseName* are included in the list.

***NumObjectsAdded***

This output parameter is an INTEGER representing the number of exclusions added to the list.

**Argument Types**

Expressions passed to this procedure must have the following data types:

- *ObjectListName*=VARCHAR
- *DatabaseName*=VARCHAR
- *ObjectName*=VARCHAR

**Usage Notes**

Caller-specified values for the input parameters *DatabaseName* and *ObjectName* may contain wildcard characters (`_`, `%`). Wildcard characters are resolved dynamically at the time the list is passed as a parameter to a called procedure. Wildcards are evaluated using standard SQL LIKE predicate semantics.

Defined object are stored in TDMaps tables *ObjectListTbl* and *ObjectListEntry*. If *AddObjectListEntrySPX* is called, the defined objects you can access are stored in TDMaps tables *ObjectListTbl* and *ObjectListEntry*.

Automatic updates are not made to existing object lists stored in TDMaps when DDL statements DROP an object.

**Examples*****Example: Creating an Object List***

In this example, you create an object list named 'MyObjects'. 'MyObjects' contains all the tables in the 'Payroll' database, the individual table 'Employee.Withholding', and all the tables in the 'Taxes' database with names that begin with 'Rate'.

```
CALL TDMaps.CreateObjectListSP('MyObjects', NULL, :ObjectListId);
CALL TDMaps.AddObjectListEntrySP('MyObjects',
'Payroll', NULL, :NumObjectsAdded);
CALL TDMaps.AddObjectListEntrySP('MyObjects',
'Employee', 'Withholding', :NumObjectsAdded);
CALL TDMaps.AddObjectListEntrySP('MyObjects',
'Taxes', 'Rate%', :NumObjectsAdded);
```

**Example: Analyzing Objects Defined in 'MyObjects'**

In this example, analyze objects defined in 'MyObjects' within all logged queries and make recommendations for moving those objects to the map 'MyObjectsMapList'.

```
CALL TDMaps.AnalyzeSP(
  'MyObjectsMapList', 'MyObjects', NULL,
  NULL, NULL, NULL, 'MyObjectsActions',
  :NumAlters, :NumExcludes, :NumLogEntries);
```

Procedure has been executed.

\*\*\* Total elapsed time was 20 minutes and 10 seconds.

NumAlters	NumExcludes	NumLogEntries
-----	-----	-----
15	1	3054

**Example: Creating Object List 'ObjNotInMap2'**

In this example, create an object list named 'ObjNotInMap2' consisting of all objects not in TD\_Map2.

```
CALL TDMaps.CreateObjectListSP('ObjNotInMap2', NULL, :ObjectListId);

INSERT TDMaps.ObjectListEntry
SELECT ZoneName, ListId, DatabaseName, TableName
FROM TDMaps.ObjectListTbl, dbc.tablesv t
WHERE ListName = 'ObjNotInMap2'
and TableKind IN ('T', 'I', 'J', 'N', 'O')
and t.mapname not in ('td_datadictionarymap', 'td_globalmap', 'td_map2');
```

**AnalyzeSP**

The AnalyzeSP procedure analyzes tables and table usage, then makes recommendations on whether to move tables to a new destination map.

The analysis is limited to a previously defined object list representing any number of databases and tables.

**ANSI Compliance**

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] AnalyzeSP (
  'DestinationMapListName',
  'ObjectListName',
  'ExclusionListName',
  [ { 'LogStartTime' | NULL } , ]
  [ { 'LogEndTime' | NULL } , ]
  { 'LogDatabase' | NULL },
  'OutputActionList',
  :NumAlters,
  :NumExcludes,
  :NumLogEntries
) [;]
```

## Syntax Elements

### **TDMaps.**

The name of the database.

### ***DestinationMapListName***

The name of an existing map list for recommending table actions on.

### ***ObjectListName***

Name of the object list. It must be specified.

### ***ExclusionListName***

Specifies the objects to skip analysis in the exclusion list.

### ***LogStartTime***

Limits query log to queries logged after a specified time.

### ***LogEndTime***

Limits query log to queries logged before a specified time.

### ***LogDatabase***

The database containing the query log tables.

### ***OutputActionList***

The name to assign to a list of table actions recommended by a procedure call.

**NumAlters**

The number of rows written to TDMaps.ActionsTbl where Action = ALTER.

**NumExcludes**

The number of rows written to TDMaps.ActionsTbl where Action = EXCLUDE.

**NumLogEntries**

The number of analyzed logged queries that reference one or more tables in the *ObjectListName*.

**Argument Types**

Expressions passed to this procedure must have the following data types:

- *DestinationMapListName*=VARCHAR
- *ObjectListName*=VARCHAR
- *ExclusionListName*=VARCHAR
- *LogStartTime*=TIMESTAMP
- *LogEndTime*=TIMESTAMP
- *LogDatabase*=VARCHAR
- *OutputActionList*=VARCHAR

**Result Type**

The result row type is:

- *NumAlters*=INTEGER
- *NumExcludes*=INTEGER
- *NumLogEntries*=BIGINT

**Usage Notes**

- You can reduce the analysis time for a query log by limiting it to queries that fall within a specified LogStartTime and/or LogEndTime. If the specified LogStartTime is NULL, the analysis reads from the start of the query log. If the specified LogEndTime is NULL, the analysis reads to the end of the query log.
- If LogDatabase is NULL, tables are grouped by database.
- For LogDatabases other than DBC and PDCRDATA, user TDMaps must be granted the SELECT privilege on its query log tables prior to calling AnalyzeSP.



- All generated actions from a call to AnalyzeSP are associated with the common identifier OutputActionList, which cannot have a NULL value. The recommended actions are written as rows to table TDMaps.ActionsTbl. Action types are limited to ALTER and EXCLUDE.
- Analysis recommendations identified by OutputActionList can be passed as an input parameter to ManageMoveTablesSP, which executes the recommended actions.
- If the OutputActionList value corresponds to an action list that already exists in the actions table, an error occurs.
- If there is no query log data, tables are grouped by database. If a log database is specified but there is no log data, then each table belongs to its own group, essentially not grouping anything.
- If the output parameter NumLogEntries returns a value of 0, the query log suggests actions to take.
- Error tables (created by CREATE ERROR TABLE) are added to ActionsTbl following the base table.

### Example: Using AnalyzeSP

To analyze tables in the Personnel object list for queries logged over the last 7 days, and then recommend actions to move them into MyNewMap:

```
CALL TDMaps.AnalyzeSP(
    'MyNewMapList', 'Personnel', NULL,
    CAST(CURRENT_TIMESTAMP - INTERVAL '7' day AS TIMESTAMP),
    CURRENT_TIMESTAMP,
    'DBC', 'MyNewMapActions', :NumAlters, :NumExcludes, :NumLogEntries);
```

Procedure has been executed.

\*\*\* Total elapsed time was 19 minutes and 47 seconds.

NumAlters	NumExcludes	NumLogEntries
-----	-----	-----
85	0	98020

### CleanUpAnalyzerSP

This procedure cleans up internal tables following a restart.

Call this procedure if the AnalyzeSP is interrupted by an abort or restart.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] CleanUpAnalyzerSP ( { 'ActionList' | NULL } ) [;]
```

## Syntax Elements

### TDMaps.

The name of the database.

### ActionList

Name of the action list to clean up. If NULL, then all rows are deleted in the ActionsTbl.

## Argument Types

The *ActionList* column for this procedure has the data type VARCHAR.

## Example: Calling CleanUpAnalyzerSP

To call the procedure:

```
CALL TDMaps.CleanUpAnalyzerSP(NULL);

Procedure has been executed.
*** Total elapsed time was 13 seconds
```

## CreateExclusionListSP

Creates a named exclusion list consisting of one or more databases, tables, join indexes, or hash indexes. These lists are useful when specifying a list of objects that the mover should not move.

This procedure works in conjunction with AddExclusionListEntrySP(X). CreateExclusionListSP is called once to create an empty list, and then AddExclusionListEntrySP(X) is called repeatedly to add objects to that list.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] CreateExclusionListSP (
    'ExclusionListName',
    [ 'DictionaryDatabase', ]
    :ExclusionListID
) [;]
```

## Syntax Elements

### **TDMaps.**

The name of the database.

### ***ExclusionListName***

Name of the exclusion list.

### ***DictionaryDatabase***

The name of the database where the data dictionary tables, such as TVM, reside.

### ***ExclusionListID***

An output parameter that returns the system-assigned numeric identifier for the list.

## Argument Types

Expressions passed to this procedure must have the following data types:

- *ExclusionListName*=VARCHAR
- *DictionaryDatabase*=VARCHAR

## Result Type

The result row type is *ExclusionListId*=BIGINT.

## Usage Notes

CreateExclusionListSP returns the system-assigned numeric identifier for the specified ExclusionListName. When calling AddExclusionListEntrySP, ExclusionListName must be specified to identify that list.

*DictionaryDatabase* defines where the data dictionary tables are stored. Either define it or use NULL. Set *DictionaryDatabase* when running AnalyzeSP on a system that is different from the system where the objects are defined. Copy the contents of the following from the source system to DictionaryDatabase on the system where AnalyzeSP and CreateExclusionListSP run:

- Maps
- Dbase
- TVM
- TVFields
- DBQLStepTbl
- ObjectUsage

- DatabaseSpace

Create the following views in DictionaryDatabase:

- Tables2V
- Databases2V
- MapsV
- QryLogStepsV
- ColumnsV
- InsertUseCountV
- UpdateUseCountV
- DeleteUseCountV
- DiskSpaceV

Finally, the rows written to ActionsTbl need to be copied to the source system.

### Example: Calling CreateExclusionListSP

To call the procedure:

```
CALL tdmmaps.CreateExclusionListSP('BillsList', NULL, :ListId);
```

## CreateExpansionMaps

Run this procedure to model map moves before a system expansion. This procedure creates three planned maps, inserts them into TDMaps.Maps, and gives them temporary names with the prefix PreExpansionMap. After the system expansion, run the PostExpansionAction procedure to rename those maps with the actual added contiguous and sparse maps. See [PostExpansionAction](#).

### ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] CreateExpansionMaps (
  'numOfNodes',
  'numOfAMPs',
  :ContiguousMapString,
  :OneAMPSParseMapString,
  :TotalNodesSparseMapString
) [;]
```

## Syntax Elements

### TD\_Maps

Input parameter. The name of the database where the procedure is located.

### *numOfNodes*

Input parameter. The number of nodes on the expanded system.

SMALLINT is the allowed data type.

### *numOfAMPs*

Input parameter. The number of AMPs on the expanded system.

SMALLINT is the allowed data type.

### *ContiguousMapString*

Output parameter. The name of the pre-expansion contiguous map.

VARCHAR is the allowed data type.

### *OneAmpSparseMapString*

Output parameter. The name of the pre-expansion 1-AMP sparse map.

VARCHAR is the allowed data type.

### *TotalNodesSparseMapString*

Output parameter. The name of the pre-expansion  $n$ -AMP sparse map, where  $n$  is the number of nodes in the system.

VARCHAR is the allowed data type.

## Usage Notes

- An error message is returned if *numOfNodes* or *numOfAMPs* is NULL.
- Use this procedure as many times as needed during expansion planning. No changes will be made to the database until the Mover procedure is run.

## Example: Using CreateExpansionMaps

In the following example, the expanded system has 4 nodes and 40 AMPs.

```
CALL TDMaps.CreateExpansionMaps(4,
                                40,
```

```

:ContiguousMapString,
:OneAmpSparseMapString,
:TotalNodesSparseMapString);

```

Result:

```

*** Procedure has been executed.
*** Total elapsed time was 1 second.
ContiguousMapString      PREEXPANSIONMAP_CONTIGUOUSMAP_4NODES
OneAmpSparseMapString    PREEXPANSIONMAP_1AMPSPARSEMAP_4NODES
TotalNodesSparseMapString PREEXPANSIONMAP_4AMPSSPARSEMAP_4NODES

```

## CreateMapListSP

Creates a named empty map list. The list of maps is a parameter of AnalyzeSP.

This procedure works in conjunction with AddMapListEntrySP. CreateMapListSP is called once to create an empty list, and then AddMapListEntrySP is called repeatedly to add individual maps to that list.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```

CALL [TDMaps.] CreateMapListSP (
  'MapListName',
  [ 'DictionaryDatabase', ]
  :MapListID
) [;]

```

## Syntax Elements

**TDMaps.**

The name of the database.

**MapListName**

Name of the map list.

**DictionaryDatabase**

The name of the database where the data dictionary tables, such as TVM, reside.

***MapListId***

An output parameter that is the system-assigned numeric identifier for the list.

**Argument Types**

Expressions passed to this procedure must have the following data types:

- *MapListName*=VARCHAR
- *DictionaryDatabase*=VARCHAR

**Result Type**

The result row type is *MapListId*=BIGINT.

**Usage Notes**

*DictionaryDatabase* defines where the data dictionary tables are stored. Either define it or use NULL. Set *DictionaryDatabase* when running AnalyzeSP on a system different from the system where the objects are defined.

Copy the contents of the following for the objects to analyze from the source system to DictionaryDatabase on the system where AnalyzeSP is run:

- Maps
- Dbase
- TVM
- TVFields
- DBQLStepTbl
- ObjectUsage
- DatabaseSpace

Create the following views in DictionaryDatabase:

- Tables2V
- Databases2V
- MapsV
- QryLogStepsV
- ColumnsV
- InsertUseCountV
- UpdateUseCountV
- DeleteUseCountV
- DiskSpaceV

### Example: Calling CreateMapListSP

To call the procedure:

```
CALL tdmmaps.CreateMapListSP('MyMapList', NULL, :ListId);
```

### CreateObjectListSP

Creates a named object list consisting of one or more databases, tables, join indexes, or hash indexes. These lists are useful when specifying large object scopes for AnalyzeSP to operate on.

This procedure works in conjunction with AddObjectListEntrySP(X). CreateObjectListSP is called once to create an empty list, and AddObjectListEntrySP(X) is called repeatedly to add individual objects to that list. A given list may be defined with any mixture of tables, join indexes, or hash indexes.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] CreateObjectListSP (
  { 'ObjectListName' | NULL },
  [ 'DictionaryDatabase', ]
  :ObjectListID
) [;]
```

### Syntax Elements

**TDMaps.**

The name of the database.

**ObjectListName**

Name of the object list. It must be specified.

**DictionaryDatabase**

The name of the database where the data dictionary tables, such as TVM, reside.

**ObjectListID**

An output parameter that is the system-assigned numeric identifier for the list.



## Argument Types

Expressions passed to this procedure must have the following data types:

- *ObjectListName*=VARCHAR
- *DictionaryDatabase*=VARCHAR

## Result Type

The result row type is *ObjectListId*=BIGINT.

## Usage Notes

CreateObjectListSP returns the system-assigned numeric identifier for the specified ObjectListName. When calling AddObjectListEntry, ObjectListName must be specified to identify that list.

*DictionaryDatabase* defines where the data dictionary tables are stored. Either define it or use NULL. Set *DictionaryDatabase* when running AnalyzeSP on a system different from the system where the objects are defined. Copy the contents of the following from the source system to DictionaryDatabase on the system where AnalyzeSP runs:

- Maps
- Dbase
- TVM
- TVFields
- DBQLStepTbl
- ObjectUsage
- DatabaseSpace

Create the following views in DictionaryDatabase:

- Tables2V
- Databases2V
- MapsV
- QryLogStepsV
- ColumnsV
- InsertUseCountV
- UpdateUseCountV
- DeleteUseCountV
- DiskSpaceV

**Example: Calling CreateObjectListSP**

To call the procedure:

```
CALL tdmaps.CreateObjectListSP('MyObjectList', NULL, :ObjectListId);
```

**MonitorAnalyzeSP**

MonitorAnalyzeSP monitors the progress of the AnalyzeSP associated with a given action list.

**ANSI Compliance**

This statement is a Teradata extension to the ANSI SQL:2011 standard.

**Syntax**

```
CALL [TDMaps.] MonitorAnalyzeSP (
  'ActionListName',
  :CompleteSteps,
  :TotalSteps
) [;]
```

**Syntax Elements****TDMaps.**

The name of the database.

**ActionListName**

Name of the analyzed action list.

**CompleteSteps**

An output parameter for the number of steps that the Analyzer finished analyzing.

**TotalSteps**

An output parameter for the total number of steps completed during analysis.

**Argument Types**

The *ActionListName* columns for this procedure has the data type VARCHAR.

## Result Types

The result row type is:

- *CompleteSteps*=INTEGER
- *TotalSteps*=INTEGER

## Example: Calling MonitorAnalyzeSP

To call the procedure:

```
CALL tdmmaps.MonitorAnalyzeSP('MyActionList', :CompleteSteps, :TotalSteps);
```

## PostExpansionAction

Run this procedure after a system expansion to rename the temporary, planned maps created with the CreateExpansionMaps procedure to the actual maps to be used in the Mover procedures. This procedure removes the prefix PreExpansionMap before the map names. This procedure is necessary if you ran the Advisor procedures before the system expansion and are not going to run them again afterward.

## ANSI Compliance

This is Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] PostExpansionAction ('numOfNodes', 'PostExpansionMap') [;]
```

## Syntax Elements

### TD\_Maps

The name of the database where the procedure is located.

### *numOfNodes*

The number of nodes in the expanded system.

SMALLINT is the allowed data type.

### *PostExpansionMap*

The name of the contiguous map after the system expansion.

VARCHAR is the allowed data type.

## Usage Notes

An error message is returned in the following cases:

- If *numOfNodes* or *PostExpansionMap* is NULL.
- If no row is found in TDMaps.Maps because the user has not run the CreateExpansionMaps procedure.
- If *numOfNodes* in the CreateExpansionMaps procedure is not the same as *numOfNodes* in the PostExpansionAction procedure.

## Example: Using PostExpansionAction

In the following example, the newly expanded system has four nodes and the contiguous map name is TD\_Map1000.

```
CALL TDMaps.PostExpansionAction(4, 'TD_Map1000');
```

Result:

```
*** Procedure has been executed.
*** Total elapsed time was 1 second.
```

The procedure substituted the planned maps created by the CreateExpansionMaps procedure with the actual maps in the Advisor procedures, as follows:

- TD\_Map1000 replaces PreExpansionMap\_ContiguousMap\_4Nodes
- TD\_1AmpSparseMap\_4Nodes replaces PreExpansionMap\_1AmpSparseMap\_4Nodes
- TD\_4AmpsSparseMap\_4Nodes replaces PreExpansionMap\_4AmpsSparseMap\_4Nodes

## Mover Procedures

The Advisor analyzes user tables and makes recommendations, such as moving a set of tables onto new maps. The Mover then acts on those recommendations, and moves the tables into the new maps.

## ManageMoveTablesSP

Moves a group of tables within a certain time limit out of the ActionsTbl and into the ActionQueueTbl queue table.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] ManageMoveTablesSP (
  [ 'JobNumber' , ]
  'SerialTableActionList' ,
  'ParallelTableActionList' ,
  { 'TimeLimit' | NULL } ,
  'GroupActions' ,
  :ActionsCompleted ,
  :NumErrors ,
  :TimeExpired
) [ ; ]
```

## Syntax Elements

### **TDMaps.**

The name of the database.

### **JobNumber**

An optional job number recorded in the history table.

### **SerialTableActionList**

Named list of table actions to perform by only one session.

### **ParallelTableActionList**

Named list of table actions to perform by one or more sessions.

### **TimeLimit**

Maximum time in minutes for executing actions. The valid range is 1 to 10080 (7 days).  
NULL indicates an unlimited duration.

### **GroupActions**

'Y' indicates related actions are scheduled and executed in a strict grouped fashion.  
'N' indicates actions within a group are scheduled and executed independently.

### **ActionsCompleted**

The number of actions that were successfully completed.

### **NumErrors**

The number of actions that encountered a failure.

***TimeExpired***

'Y' indicates the time limit was exceeded prior to completing all actions in TableActionList. Otherwise, the parameter is 'N'.

**Argument Types**

Expressions passed to this procedure must have the following data types:

- *JobNumber*=INTEGER
- *SerialTableActionList*=VARCHAR
- *ParallelTableActionList*=VARCHAR
- *TimeLimit*=INTEGER
- *GroupActions*=CHAR

**Result Type**

The result row type is:

- *ActionsCompleted*=INTEGER
- *NumErrors*=INTEGER
- *TimeExpired*=CHAR

**Usage Notes**

All actions stored within TDMaps.ActionsTbl that are part of the specified SerialTableActionList or ParallelTableActionList and whose type are not 'Exclude' are fetched and queued for execution in a group at a time. Objects in the SerialTableActionList are processed by only one session. Objects in ParallelTableActionList are processed by one or more sessions. One or both parameters must be specified. If both are specified, one session works on the SerialTableActionList and the other session(s) work on the ParallelTableActionList. After all the not 'Exclude' actions are processed, then 'Exclude' actions are moved to the history table without submitting an ALTER TABLE.

Generate table action lists from prior calls to TDMaps.AnalyzeSP. The value specified for AnalyzeSP parameter OutputActionList must be the same value specified for parameter SerialTableActionList or ParallelTableActionList. Prior to calling ManageMoveTablesSP, you can customize the actions generated in OutputActionList by executing standard UPDATE, DELETE, and INSERT statements on TDMaps.ActionsTbl.

Prior to being queued for execution, all qualifying actions are grouped and ordered according to the designated column values in ActionsTbl.GroupOrder and ActionsTbl.ActionOrder. Sets of tables are assigned to the same group as determined by prior calls to procedure AnalyzeSP, if the Analyzer is run with the log data and determines there is a join relationship.

After calling `ManageMoveTablesSP`, one or more concurrent calls must be made to related procedure `MoveTablesSP` to execute the queued actions. Because a call to `ManageMoveTablesSP` does not complete and return until all actions complete (or `TimeLimit` expires), separate DBS sessions are required to issue calls to `MoveTablesSP`.

Only one active execution of `ManageMoveTablesSP` is allowed in the system. The underlying system queue tables used for scheduling and executing actions do not support more than one `SerialTableActionList` or `ParallelTableActionList` at a given time.

Results from executing this procedure and the worker procedures it manages are written to table `TDMaps.ActionHistoryTbl` whose rows represent completed or in progress actions. Monitor and display results by issuing standard `SELECT` statements on the table.

If the specified `TimeLimit` expires before all actions in `SerialTableActionList` or `ParallelTableActionList` complete, no additional groups of actions queue for execution.

The default behavior is to manage actions with the same group as a single unit with respect to scheduling, executing, and time limit expiration processing. Override the behavior by specifying input parameter `GroupActions` as 'N'. The grouping behavior ensures that `Alter` (move) actions on groups of tables with strong joining relationships complete by a given call to `ManageMoveTablesSP`. If the Analyzer is run without log data, then tables are grouped by database.

If `GroupActions` is 'Y' (or `NULL`) and a non-`NULL` `TimeLimit` is specified, `ManageMoveTablesSP` cannot schedule a group of actions to execute if their combined estimated time of completion exceeds the current time remaining. Similarly, when `GroupActions` is 'N' and a non-`NULL` `TimeLimit` is specified, `ManageMoveTablesSP` does not execute the action for a given table if its estimated time of completion exceeds the current time remaining. When this occurs for a given group or table whose assigned `GroupOrder` or `ActionOrder` will execute, it is recorded within `ActionHistoryTbl` with a `Status` value of `Skipped`.

If `TimeLimit` expires before all actions within a given group complete, all actions in that group complete before stopping execution.

When `GroupActions` is 'N', and the time limit expires, any queued actions in `TDMaps.ActionQueueTbl` that did not run are transferred back to table `TDMaps.ActionsTbl`. They are assigned a `GroupOrder` representing the highest priority within their designated `ActionListName`.

If the returned value of output parameter `TimeExpired` is 'Y', complete any remaining actions by calling `ManageMoveTablesSP` again with the same `TableActionList`. `ManageMoveTablesSP` automatically resumes the uncompleted actions in the list.

To stop a running call to `ManageMoveTablesSP`, call procedure `StopMoveTablesSP`.

### Example: Executing the Actions in the 'MyNewMapActions' List

In this example, execute the actions in the 'MyNewMapActions' list and impose a time limit of 12 hours. Then, invoke two concurrent worker procedures to perform the actions.

Session 1:

```
CALL TDMaps.ManageMoveTablesSP(NULL, NULL, 'MyNewMapActions', 720, 'Y',
    :ActionsCompleted, :NumErrors, :TimeExpired);
```

Procedure has been executed.

\*\*\* Total elapsed time was 11 hours 19 minutes and 49 seconds.

ActionsCompleted	NumErrors	TimeExpired
-----	-----	-----
85	0	'N'

Session 2:

```
CALL TDMaps.MoveTablesSP('P',NULL,NULL,NULL,NULL);
```

Procedure has been executed.

\*\*\* Total elapsed time was 11 hours 3 minutes and 5 seconds.

Session #3:

```
CALL TDMaps.MoveTablesSP('P',NULL,NULL,NULL,NULL);
```

Procedure has been executed.

\*\*\* Total elapsed time was 11 hours 19 minutes and 45 seconds.

## MoveTablesSP

MoveTablesSP executes a worker session that consumes and executes actions from table ActionQueueTbl, which was populated by a prior call to ManageMoveTablesSP. You can make multiple concurrent calls to MoveTableSP, where the workers fetch and perform actions in parallel from the shared queue.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] MoveTablesSP (
    'ActionList',
    'DatabaseName',
    'TableName',
    'TargetMapName',
    'ColocationName'
) [;]
```



## Syntax Elements

### **TDMaps.**

The name of the database.

### **ActionList**

The options are 'P' for the ParallelTableActionList, or 'S' for the SerialTableActionList.

### **DatabaseName**

The database in which the object is defined.

### **TableName**

Name of table, join index, or hash index defined in *DatabaseName*.

### **TargetMapName**

Name of the map where the object is moved.

### **ColocationName**

The colocation name string for the table.

## Argument Types

Expressions passed to this procedure must have the following data types:

- *ActionList*=CHAR
- *DatabaseName*=VARCHAR
- *TableName*=VARCHAR
- *TargetMapName*=VARCHAR

## Usage Notes

ActionList is 'P' for the procedure to work on the ParallelTableActionList and 'S' for the procedure to work on the SerialTableActionList. If specified, DatabaseName, TableName, and TargetMapName must be NULL.

DatabaseName, TableName, and TargetMapName identify the object to move and the map to move it to. If specified, ActionList must be NULL.

This procedure cannot move objects in the TDMaps database.

Moving an object to a new map via the ALTER statement requires the DROP TABLE privilege on the object being moved. Moving an object to a new map via CALL MoveTablesSP does not require the DROP TABLE privilege on the object being moved.

Use a separate DBS session for each concurrent call to MoveTablesSP. Use BTEQ commands SET SESSIONS and REPEAT.

A call to MoveTablesSP completes when there are no more actions in the queue to complete or the time limit specified in the call to ManageMoveTableSP expires.

MoveTablesSP does not return output parameters summarizing the status of its completed actions. Instead, the combined status of all actions performed by all calls to MoveTableSP is reported in the results returned by ManageMoveTablesSP.

Results from executing the procedure are written to table TDMaps.ActionHistoryTbl whose rows represent completed or in progress actions.

Callers who wish to stop a running call to MoveTablesSP must call procedure StopMoveTablesSP rather than aborting the procedure. If the procedure is aborted or a restart occurs, then tables being moved have an “InProgress” status. Calling CleanUpMoveTablesSP changes the status to “Aborted”. You can also call TDMaps.StopMoverSP, which takes an integer as input representing the number of Mover tasks to stop.

The maximum number of mover sessions is 128.

## Examples

### *Example: Using BTEQ to Execute Queued Actions*

Using BTEQ, execute any queued actions from a prior outstanding call to ManageMoveTablesSP.

```
BTEQ -- Enter your SQL request or BTEQ command:
.SET SESSIONS 3
.LOGON machine/user,passwd
.REPEAT 3
CALL TDMaps.MoveTablesSP('P',NULL,NULL,NULL,NULL);
*** Ok, Session 32015, Request 3, Statement# 1
*** Procedure has been executed.
*** Ok, Session 32016, Request 3, Statement# 1
*** Procedure has been executed.
*** Ok, Session 32017, Request 3, Statement# 1
*** Procedure has been executed.
*** Total number of statements: 3, Accepted: 3, Rejected: 0
*** Total elapsed time was 3 hours.
*** Total requests sent to the DBC = 3
*** Successful requests per hour = 1.000
```

**Example: Move a Table to a Map**

Using BTEQ, move the EmployeeTbl table to the map TD\_Map2.

```
BTEQ -- Enter your SQL request or BTEQ command:
.LOGON machine/user,passwd
CALL
TDMaps.MoveTablesSP(NULL,'HR','EmployeeTbl','TD_Map2','MyColocationName');
*** Ok
*** Procedure has been executed.
```

**MonitorMoveTablesSP****ANSI Compliance**

This statement is a Teradata extension to the ANSI SQL:2011 standard.

**Syntax**

```
CALL [TDMaps.] MonitorMoveTablesSP (
  'ActionListName',
  :NumTables,
  :NumComplete,
  :NumInProgress,
  :NumWaitingToStart,
  :PercentComplete
) [;]
```

**Syntax Elements****TDMaps.**

The name of the database.

**ActionListName**

The name of the action list to get status on.

**NumTables**

The total number of objects to move to the new map.

**NumComplete**

The number of objects already moved to the new map.

**NumInProgress**

The number of objects currently moving to the new map.

**NumWaitingToStart**

The number of objects currently waiting to move.

**PercentComplete**

The percent of objects already moved to the new map.

**Argument Types**

Expressions passed to this procedure have the *ActionListName*=VARCHAR data type.

**Result Type**

The result row type is:

- *NumTables*=INTEGER
- *NumComplete*=INTEGER
- *NumInProgress*=INTEGER
- *NumWaitingToStart*=INTEGER
- *PercentComplete*=DECIMAL

**Example: Using MonitorMoveTablesSP**

```
CALL TDMaps.MonitorMoveTablesSP('MyNewMapActions',NumTables,
                                NumComplete, NumInProgress,
                                NumWaitingToStart, PercentComplete);
```

```
*** Procedure has been executed.
```

```
*** Warning: 3212 The stored procedure returned one or more result sets.
```

```
*** Total elapsed time was 1 second.
```

NumTables	2712
NumComplete	2526
NumInProgress	1
NumWaitingToStart	185
PercentComplete	.93

```
*** ResultSet# 1 : 1 rows returned by "TDMAPS.MONITORMOVETABLESSP".
```

```

ZoneName ?
JobNumber      2016042801
Action Alter
Status InProgress
DatabaseName MHM
TableName TAB21_PI
GroupOrder    1.00
SourceMap TD_Map1
DestinationMap TD_Map2
ActionSQLText ALTER TABLE MHM.TAB21_PI, MAP=td_map2
ActionListName MyNewMapActions
TableSize      1118482
FractionOfPermDBFree 9.99999999610461E-001
Issues N
PrevGroupsBytesPerSec ?
EstElapsedTimeInSecond ?
ElapsedTime ?
StartTime 2016-02-02 12:22:45.270000+00:00
EndTime ?
ErrorCode      0
RunUID 0000F903
ActionsTblDescription Move table for system expansion.
Description ?
WorkerId      1
--

```

## StopMoveTablesSP

Stops queuing and executing additional action groups using a running call to ManageMoveTablesSP.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```

CALL [TDMaps.] StopMoveTablesSP (
  'StopQueuedActions',
  :NumActionsStopped,
  :NumQueuedActionsStopped
) [;]

```

## Syntax Elements

### **TDMaps.**

The name of the database.

### ***StopQueuedActions***

'N' ensures all actions within the queued group complete before terminating.

'Y' allows for stopping individual queued actions that have yet to run.

### ***NumActionsStopped***

The number of remaining actions in TableActionList that were stopped before queueing for execution.

### ***NumQueuedActionsStopped***

The number of already-queued actions that were stopped before executing. Applicable only when *StopQueuedActions* is 'Y'.

## Argument Types

Expressions passed to this procedure have the *StopQueuedActions*=CHAR data type.

## Result Type

The result row type is:

- *NumActionsStopped*=INTEGER
- *NumQueuedActionsStopped*=INTEGER

## Usage Notes

To ensure that all related queued actions for a group are completed by a given call to ManageMoveTablesSP, specify a value of 'N' or NULL for the parameter StopQueuedActions. If not, related tables reside in different maps and require additional data redistribution steps when running queries on those tables.

When the StopQueuedActions specified value is 'Y', any queued actions in TDMaps.ActionQueueTbl are stopped and transferred back to table TDMaps.ActionsTbl. The actions are assigned a GroupOrder representing the highest priority within the designated ActionListName. You can reschedule the actions later by calling ManageMoveTablesSP.

Any action types 'Alter' already running are not aborted.

Complete any stopped actions at a later time by calling `ManageMoveTablesSP` with the same `TableActionList`. `ManageMoveTablesSP` automatically resumes the execution of stopped and pending actions in the list.

Use `ManageMoveTablesSP` to specify a value for its `TimeLimit` parameter to eliminate calling `StopMoveTablesSP` later. `StopMoveTablesSP` uses the same logic as the stop logic when a time limit expires.

Queued actions that are stopped are recorded as events within table `ActionHistoryTbl` with a value of 'Stopped' in the `Status` column. Unqueued items are not recorded in `ActionHistoryTbl`.

### Example: Using `StopMoveTablesSP`

```
CALL TDMaps.StopMoveTablesSP('N',
                             :NumActionsStopped, :NumQueuedActionsStopped);
```

Procedure has been executed.

\*\*\* Total elapsed time was 2 minutes 3 seconds

NumActionsStopped	NumQueuedActionsStopped
-----	-----
25	0

## StopMoverSP

Stops one or more `MoveTablesSP` procedures.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] StopMoverSP ('NumSPToStop') [;]
```

### Syntax Elements

**TDMaps.**

The name of the database.

***NumSPToStop***

The number of `MoveTablesSP` procedures to stop.

## Argument Types

Expressions passed to this procedure have the *NumSPToStop*=INTEGER data type.

### Example: Using StopMoverSP

To call the procedure:

```
CALL TDMaps.StopMoverSP(2);

Procedure has been executed.
*** Total elapsed time was 2 minutes 3 seconds
```

## StopSerialWorkerSP

Stops a MoveTablesSP procedure serial worker session.

### ANSI Compliance

This is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] StopSerialWorkerSP () [;]
```

### Syntax Elements

**TDMaps.**

The name of the database where the procedure is located.

### Usage Notes

This procedure stops a hanging serial worker session. Use this procedure if the manager session is not present and the serial worker session is still active.

### Example: Using StopSerialWorkerSP

To call the procedure:

```
CALL TDMaps.StopSerialWorkerSP();
```



## CleanUpMoveTablesSP

Cleans up internal tables following a restart and changes InProgress status to Aborted in the history table.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
CALL [TDMaps.] CleanUpMoveTablesSP () [;]
```

### Syntax Elements

#### TDMaps.

The name of the database.

### Usage Notes

CleanUpMoveTablesSP cleans the following list of tables:

- TDMaps.ManagerSPInstanceTbl
- TDMaps.WorkerSPInstanceTbl
- TDMaps.ActionQueueTbl
- TDMaps.SerialActionQueueTbl
- TDMaps.EndOfGroupQueueTbl
- TDMaps.StopMoveQueueTbl

### Example: Calling CleanUpMoveTablesSP

To call the procedure:

```
CALL TDMaps.CleanUpMoveTablesSP();
```

```
Procedure has been executed.
```

```
*** Total elapsed time was 13 seconds
```

## MoveTDMapsTablesSP

Moves tables in TDMaps to a target map.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
CALL [TDMaps.] MoveTDMapsTablesSP (
  'DestinationMap',
  :NumInTotal,
  :NumComplete,
  :NumFailed
) [;]
```

## Syntax Elements

### TDMaps.

The name of the database.

### *DestinationMap*

The target map to move TDMaps tables.

### *NumInTotal*

The total numbers of tables in TDMaps to move.

### *NumComplete*

The number of tables in TDMaps that were moved successfully.

### *NumFailed*

The number of tables in TDMaps that did not move.

## Argument Types

The DestinationMap expression passed to this procedure has the data type VARCHAR.

## Result Type

The result row type is:

- *NumInTotal*=INTEGER
- *NumComplete*=INTEGER
- *NumFailed*=INTEGER

### Example: Calling MoveTDMapsTablesSP

To call the procedure:

```
CALL
tdmaps.MoveTDMapsTablesSP('TargetMap', :NumInTotal, :NumComplete, :NumFailed);
```

### TruncateHistorySP

Truncates old history rows from TDMaps tables. Periodically run this procedure to prevent historical data from consuming an excessive amount of space.

#### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

#### Syntax

```
CALL [TDMaps.] TruncateHistorySP ('OlderThan', :NumDeletedRows) [;]
```

#### Syntax Elements

##### TDMaps.

The name of the database.

##### OlderThan

Deletes TDMaps history data older than the specified time. It cannot be NULL.

##### NumDeletedRows

The output parameter for the number of deleted rows in ActionHistoryTbl, LogTbl, and AnalyzeLogTbl.

#### Argument Types

The *OlderThan* expression passed to this procedure has the data type `TIMESTAMP`.

#### Result Type

The result row type is *NumDeletedRows*=`INTEGER`.

## Usage Notes

The contents of ActionHistoryTbl represent a historical log of map-related actions and events. You can safely remove the contents without significantly impacting map management operations.

### Example: Calling TruncateHistorySP

To call the procedure:

```
CALL TDMaps.TruncateHistorySP(current_timestamp, :NumofDeleteRows);
*** Procedure has been executed.
*** Warning: 3212 The stored procedure returned one or more result sets.
*** Total elapsed time was 2 seconds.
```

NumDeletedRows

```
-----
1
```

\*\*\* ResultSet# 1 : 3 rows returned by "TDMAPS.TRUNCATEHISTORYSP".

Table	NumRowsDeleted
-----	-----
ActionHistory	0
LogTbl	1
AnalyzeLogTbl	0

# Table Operators

Table operators accept a table or table expression as input and generate a table as output. They are a type of user-defined function (UDF). Table operators are used only in the FROM clause of SELECT statements.

Table operators enable a simplified in-database MapReduce-style programming model.

Scalar sub-queries (SSQ) are sub-queries that result in a single value. SSQ is not supported in table operators with multiple ON clauses or ON clauses using PARTITION BY or HASH BY.

---

## Note:

You can run table operators on a specific set of AMPs, such as hardware with different connectivity options to access the database. Create a contiguous map to include the nodes, or connectivity options to access the database or different performance characteristics. Then, run a CREATE/REPLACE FUNCTION or CREATE FOREIGN SERVER statement to reference the new contiguous or sparse map.

---

## CALCMATRIX

Calculates a sum-of-squares-and-cross-products (SSCP) matrix.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Required Privileges

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

### Syntax

```
[TD_SYSFNLIB.] CALCMATRIX (
  ON { tableName | ( query_expression ) }
  [ HASH BY hashByColList [,...] ]
  [ LOCAL ORDER BY localOrderByListSpec [,...] ]
  USING PHRASE ('value')
  [ CALCTYPE ('value') ]
  [ OUTPUT ('value') ]
  [ NULL_HANDLING ('value') ]
)
```

## Syntax Elements

### *localOrderByListSpec*

```
localOrderByList [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

### **TD\_SYSFNLIB.**

Name of the database where the function is located.

### ***tableName***

The name of the table with the input data for the function.

### ***query\_expression***

The SELECT statement with input data for the function.

### **hashByColList**

Specifies the column name set by which rows are to be hashed across the AMPs before being passed to the operator.

### **localOrderByList**

Specifies the column name set by which rows are to be value-ordered locally on the AMPs before being passed to the operator.

### **ASC**

Results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

### **DESC**

Results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

### **NULLS FIRST**

NULL results are to be listed first.

**NULLS LAST**

NULL results are to be listed last.

**PHASE**

Specifies one of the phases of the function. Valid values:

- 'LOCAL'
- 'COMBINE'

For a definition of LOCAL and COMBINE, see [Usage Notes](#).

**CALCTYPE**

Specifies the type of matrix CALCMATRIX creates. Valid values:

- 'SSCP' = sum-of-squares-and-cross-products matrix (the default)
- 'ESSCP' = Extended-sum-of-squares-and-cross-products matrix
- 'CSSCP' = Corrected-sum-of-squares-and-cross-products matrix
- 'COV' = Covariance matrix
- 'COR' = Correlation matrix

**OUTPUT**

Specifies the format of the output. Valid values:

- 'COLUMNS'
- 'VARBYTE'

**NULL\_HANDLING**

Specifies how nulls that appear as in any data columns are handled. Valid values:

- 'ZERO' = any NULL values are considered to be 0, and the row is processed (default).
- 'IGNORE' = the row is skipped.

**Data Type**

The data type of the data columns must be some numeric data type.

## Usage Notes

### Note:

CALCMATRIX processes data representing a single input matrix or multiple input matrices. In the latter case, the input must have column(s) that identify which matrix a given input row belongs to. These columns are called as *matrix id columns* and the value of one tuple of matrix id columns is called a *matrix id*. All rows from the same matrix must have the same matrix id.

CALCMATRIX works in LOCAL and COMBINE phases.

- In the LOCAL phase, the function processes the input data that exists in the source table on each amp separately and outputs rows of summarized data for that amp. No data has to be moved among amps.
- In the COMBINE phase, the function processes the summarized data from each amp (that is, the output from the LOCAL phase) and combines it to form the rows that make up final output.

In order to get one result matrix (or, if there are matrix id columns, one result matrix for each matrix id), the COMBINE phase must use a HASH BY clause and possibly a LOCAL ORDER BY clause.

### LOCAL Phase Details

- The input to the LOCAL phase must consist of zero or more matrix id columns, and 1 or more data columns (call this number *n*). The matrix id columns, if present, must be placed in the LOCAL ORDER BY list.
- The output from the LOCAL phase consists of the matrix id columns, if any, an INT column named *rownum*, a VARCHAR(128) column named *rowname*, a BIGINT column named *c* (for count), a REAL column named *s* (for sum), and a REAL column for each data column of input.
- The CALCTYPE and OUTPUT custom values in the LOCAL phase are always set to 'SSCP' and 'COLUMNS' respectively. Attempts to use different values for these keys are ignored.
- If any of the data columns in a row are NULL, the row's handling depends on the setting for NULL\_HANDLING.

### COMBINE Phase Details

- The input to the COMBINE phase must be exactly the columns that are output from the LOCAL phase: zero or more matrix id columns, an INT *rownum* column, a VARCHAR(128) *rowname* column, a BIGINT column named *c* (for count), a REAL column names *s* (for sum), and a REAL column for each data column of input.

If there are matrix id columns in the data, they should be specified in the HASH BY and LOCAL ORDER BY lists. If not, a column with a constant value and an alias name, like "1 as p", should be added to the ON clause and the HASH BY list.

- The custom values for CALCTYPE and OUTPUT control the output columns from the COMBINE phase.
- If the OUTPUT is set to 'COLUMNS', then the columns output from the 'COMBINE' phase are: The matrix id columns, if present, an INT *rownum* column, a VARCHAR(128) *rowname* column, a REAL



column for each data column of input and, if the CALCTYPE is 'ESSCP', a BIGINT column named *c* (for count), a REAL column named *s* (for sum).

If the OUTPUT is set to 'VARBYTE', then the columns output from the 'COMBINE' phase are: an INT *rowname* column, a VARCHAR(128) *rowname* column, and a VARBYTE column named *v*.

- If there are no matrix id columns, the final output of the COMBINE phase consists of *n* rows, where *n* is the number of data columns in the input. The rownum field has values from 1 to *n*. Thus each output row is one row from the resulting matrix. If there are matrix id columns, the final output of the COMBINE phase consists of *n* rows per matrix id. The rownum field will have values from 1 to *n*.

## VARBYTE Output

When the OUTPUT is 'VARBYTE', the output row consists of the matrix id columns, if any, and these fields:

- INT rownum
- VARCHAR(128) rowname
- VARBYTE *value*

## Details for Both the LOCAL and the COMBINE Phases

- The values for the *rownum* column ranges from 1 to *n*, the number of data columns in the input. If the *rownum* column has value *x*, the *rowname* column is the *x*'th data column name.
- If the columns in the ON clause are expressions, they must have an alias name. Columns in the LOCAL ORDER BY or HASH BY lists must be column or alias names from the ON clause, or ordinal values.

## Examples

### Example: One Input Matrix, Single SQL Statement

In the following example, table T has 1 billion rows of weather data. It has 1 geometry column representing the point (latitude/longitude) of the observation, and these REAL columns:

- temperature
- air\_pressure
- rainfall.

Use an SQL statement that invokes CALCMATRIX twice, once for the LOCAL phase and once for the COMBINE phase, as follows:

```
select * from CALCMATRIX( ON (
  select 1 as p, X.* from CALCMATRIX( ON (select temperature,
air_pressure, rainfall from T)
  USING PHASE('LOCAL') )X )
  HASH BY p
```

```

USING PHASE('COMBINE')
)Y;

```

The result set would be the SSCP matrix for the weather data. The following table shows the result set. *n* means the numeric result of the calculation.

rownum	rowname	temperature	air_pressure	rainfall
1	temperature	<i>n</i>	<i>n</i>	<i>n</i>
2	air_pressure	<i>n</i>	<i>n</i>	<i>n</i>
3	rainfall	<i>n</i>	<i>n</i>	<i>n</i>

## Example: One Input Matrix, Multiple SQL Statements

In the following example, table T has the same weather data as table T in “Example: One Input Matrix, Single SQL Statement.” However, the aim this time is to produce both the covariance and correlation matrices. In order to avoid 2 passes over billion rows of raw data, insert the results of CALCMATRIX using the LOCAL phase into an intermediate table. Then run CALCMATRIX using the COMBINE phase twice, but against the intermediate table, which only has 3 \* (the number of amps in the system) rows.

```

CREATE TABLE T_INTERMEDIATE as (
  select 1 as p, X.* from CALCMATRIX( ON (select temperature,
air_pressure, rainfall from T) USING PHASE('LOCAL') )X
) WITH DATA;
select * from CALCMATRIX ( ON T_INTERMEDIATE
  HASH BY p
  USING PHASE('COMBINE') CALCTYPE('COV')
)Y;
select * from CALCMATRIX( ON T_INTERMEDIATE
  HASH BY p
  USING PHASE('COMBINE') CALCTYPE('COR')
)Y;

```

## Example: Multiple Input Matrices, One SQL Statement

In the following example, table T has the same weather data as table T in [Example: One Input Matrix, Single SQL Statement](#) and [Example: One Input Matrix, Multiple SQL Statements](#). In this example, in addition to the point location of the observation, there is an additional VARCHAR column for continent, which is the matrix id. Our aim is to produce 7 SSCP matrices, one for each continent's data.

Use an SQL statement that invokes CALCMATRIX twice, once for the LOCAL phase and once for the COMBINE phase.

```

select * from calcmatrix( ON (
    select * from calcmatrix( ON (select continent, temperature,
air_pressure, rainfall from T) LOCAL ORDER BY continent
USING    PHASE('LOCAL') )X )
    HASH BY continent
    LOCAL ORDER BY continent
    USING PHASE('COMBINE')
    )Y
    order by continent, rownum;

```

The result set in the following tables is the SSCP matrix for weather data by continent.

**Note:**

Only 2 of the matrices are shown below.

continent	rownum	rowname	temperature	air_pressure	rainfall
Europe	1	temperature	<i>n</i>	<i>n</i>	<i>n</i>
Europe	2	air_pressure	<i>n</i>	<i>n</i>	<i>n</i>
Europe	3	rainfall	<i>n</i>	<i>n</i>	<i>n</i>

continent	rownum	rowname	temperature	air_pressure	rainfall
Asia	1	temperature	<i>n</i>	<i>n</i>	<i>n</i>
Asia	2	air_pressure	<i>n</i>	<i>n</i>	<i>n</i>
Asia	3	rainfall	<i>n</i>	<i>n</i>	<i>n</i>

## Related Information

For more information on activating and invoking embedded services functions, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Cogroups

Cogroups are used with protected and non-protected C/C++ and Java table operators.

Cogroups are used for table operators that contain multiple ON clauses. By default, the input from the ON clauses is pooled together into one group as long as the input has the same value as the partition key. Input that does not qualify is ignored to ensure that the results make sense.

The cogroup feature is on by default, but if necessary, you can turn off this functionality by calling `FNC_TblOpDisableCoGroup()` in the contract function.

## Example: Cogroup Used for a Table Operator with Multiple Inputs

This example demonstrates the grouping operation used with multiple inputs to a table operator, called a cogroup. All of the inputs that satisfy the condition are grouped together in a single group. Cogroup matches the partition keys of multiple inputs, thus ensuring that the results are correct. Using a cogroup enhances the ability of table operators to work with multiple input streams and dimension tables.

Suppose that you have the following two tables.

WebLog		
cookie	cart_amount	page
AAAA	\$60	Thankyou
AAAA	\$140	Thankyou
BBBB	\$100	Thankyou
CCCC		Intro
CCCC	\$200	Thankyou
DDDD	\$100	Thankyou

AdLog		
cookie	ad_name	action
AAAA	Champs	Impression
AAAA	Puppies	Click
BBBB	Apples	Click
CCCC	Baseball	Impression
CCCC	Apples	Click

You run the following query:

```
SELECT cookie, cart_amt, adname, action
  FROM attribute_sales (
    ON (SELECT cookie, cart_amt FROM weblog
        WHERE page = 'thankyou' ) as W PARTITION BY cookie
    ON adlog as S PARTITION BY cookie) as result1 ;
```

The inputs are grouped together similar to the following table.

Grouped and Nested Relations With the Cogroup on Cookie		
Cookie	WebLog	AdLog
AAAA	AAAA,\$60,thankyou AAAA,\$140,thankyou	AAAA,champs,impression AAAA,puppies,click
BBBB	BBBB,\$100,thankyou	BBBB,apples,click
CCCC	CCCC,\$200,thankyou	CCCC,baseball,impression CCCC,apples,click
DDDD	DDDD,\$100,thankyou	

The output of the query is similar to the following.

adname	attr_revenue
champs	\$40
puppies	\$160
apples	\$240
baseball	\$40

Each time you invoke the table operator, only the inputs that have the same partition key values participate in the invocation.

If there are no rows of the PARTITION BY ANY and PARTITION BY key inputs on an AMP and the query involves DIMENSION input, the table operator is not invoked even though the DIMENSION table has rows on that AMP.

## Related Information

- For more information about FNC calls, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For information about the use of multiple ON clauses, see *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

## FeatureNames\_TBF

Use the FeatureNames\_TBF table function to perform an easy lookup when aggregating feature use counts.

Table function FeatureNames\_TBF was added as part of the Feature Use Logging feature, and is created automatically. The table helps answer the following queries when joined with the DBC.DBQLogTbl.FeatureUsage column using BIT manipulation functions:

- How many times was a particular feature used in the past week?

- How many times was a feature used per day?
- How many times was a feature used in a given period of time?
- What is the most-used feature in Vantage?
- What percentage of the requests used a certain feature?
- When was the last time a particular feature was used?

For example, to find what percentage of the requests in DBC.DBQLogTbl use 'Block Level Compression,' use:

```
SELECT CAST((b.FeatureName as char(50)),
cast( cast(NULLIFZERO(sum(GetBit(a.FeatureUsage,(2047 - b.FeatureBitpos)))) as
FLOAT)/count(a.QueryID)*100 as FLOAT FORMAT '----,---,---,---,--9.999' ) as
FeatureUsePercent from DBC.dbqlogtbl a, DBC.QryLogFeatureListV b group by
b.FeatureName where b.FeatureName = 'Block Level Compression';
```

FeatureName	FeatureUsePercent
-----	-----
Block Level Compression	65.164

To find the usage percentage of all the features, use:

```
SELECT CAST(b.FeatureName as char(50)),
cast( cast(NULLIFZERO(sum(GetBit(a.FeatureUsage,(2047 - b.FeatureBitpos)))) as
FLOAT)/count(a.QueryID)*100 as FLOAT FORMAT '----,---,---,---,--9.999' ) as
FeatureUsePercent from DBC.dbqlogtbl a, DBC.QryLogFeatureListV b group by
b.FeatureName order by 2 desc;
```

## Syntax

```
[SYSLIB.] FeatureNames_TBF ( )
```

## Syntax Elements

### SYSLIB.

Name of the database where the function is located.

## Columns

The FeatureNames\_TBF table function contains the following columns:

Column Name	Type	Description
FeatureBitPos	INTEGER	Represents the bit position on the DBC.DBQLogTbl. FeatureUsage column.
FeatureName	VARCHAR(100)	Holds the name of feature represented by the bit.

## Example: Generating a Report on Feature Usage

The following sample query generates a report on feature usage based on all the DBQLogTbl rows using the FeatureUsage bitmap.

```
select cast(b.FeatureName as char(50)) , sum(GetBit(a.FeatureUsage,(2047
- b.FeatureBitpos))) as FeatureuseCount from DBC.DbqLogTbl a,
DBC.QryLogFeatureListV b group by b.FeatureName;
```

An example of a query return output:

FeatureName	FeatureuseCount
Character Partition Primary Index	4
Multi Level Partition Primary Index	52
Increased Partition Level Partition Primary Index	1
Partition Primary Index	65

## R Table Operator

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.

### Note:

The ExecR table operator is unavailable by default to users of Teradata Vantage delivered as-a-service, such as on AWS and Azure. Contact your Teradata account representative to have it enabled.

## Installation of R Components and Packages

R table operators require the following components:

- The R interpreter
- The udfGPL library
- The TDR R package

Your system administrator must install these components before you can use R functionality with Teradata. In addition, you must run the optional DIP script `DipRTblOp` to create the ExecR table operator, which will be used to execute R scripts.

## Using ExecR to Run 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 run 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 run 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_clause ]
  [ PARTITION_BY_clause ]
  [ LOCAL_ORDER_BY_clause ]
  [ DIMENSION_clause ]
  [ ON tab2 ]...
  USING
  Contract ('R_contract_function')
  Operator ('R_operator')
  [ other_USING_clause ]...
) 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_clause ]
  [ PARTITION_BY_clause ]
  [ LOCAL_ORDER_BY_clause ]
  [ DIMENSION_clause ]
```



```
[ ON tab2 ]...
USING
Contract (SELECT rcode FROM prohtable WHERE id=1)
Operator (SELECT rcode FROM prohtable WHERE id=2)
[ other_USING_clause ]...
) 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. 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;
```

## Related Information

- For more information about the R table operator, see *Teradata Vantage™ - SQL External Routine Programming, B035-1147*.
- For instructions on installing the R table operator components, see *Teradata Vantage™ - SQL External Routine Programming, B035-1147*.

## READ\_NOS

Access external files in JSON, CSV, or Parquet format.

### Privileges

You must have the EXECUTE FUNCTION privilege on TD\_SYSFNLIB.READ\_NOS.

## READ\_NOS Syntax

```
READ_NOS (
[ ON { table_name | view_name | ( query_expression ) } ]
USING (
  LOCATION ( 'external_file_path' )
  [ AUTHORIZATION ( { [DatabaseName.]AuthorizationObjectName |
    '{ "Access_ID": "identification", "Access_Key": "secret_key" }' } ) ]
  [ BUFFERSIZE ( 'buffer_size' ) ]
  [ RETURNTYPE ( { 'NOSREAD_RECORD' | 'NOSREAD_KEYS' |
    'NOSREAD_PARQUET_SCHEMA' 'NOSREAD_SCHEMA' } ) ]
  [ SAMPLE_PERC ( 'row_sampling_value' ) ]
```

```
[ STOREDAS ( { 'PARQUET' | 'TEXTFILE' } ) ]
[ FULLSCAN ( { 'TRUE' | 'FALSE' } ) ]
[ MANIFEST ( { 'TRUE' | 'FALSE' } ) ]
[ ROWFORMAT ( 'rowformat_value' ) ]
[ HEADER ( { 'TRUE' | 'FALSE' } ) ]
)
```

**Note:**

You must type the bold curly braces shown in `'{"Access_ID": "identification", "Access_Key": "secret_key"}'`. Those particular curly braces are part of the READ\_NOS statement.

## READ\_NOS Syntax Elements

***table\_name***

Name of table.

***view\_name***

Name of view.

***query\_expression***

Expression to specify input to table operator.

**LOCATION**

You must specify a LOCATION value, which is a Uniform Resource Identifier (URI) pointing to the data in the external object storage system. The LOCATION value includes the following components:

- **Amazon S3:** `/connector/bucket.endpoint/[key_prefix]`
- **Azure Blob storage and Azure Data Lake Storage Gen2:** `/connector/storage-account.endpoint/container/[key_prefix]`
- **Google Cloud Storage (GCS):** `/connector/endpoint/bucket/[key_prefix]`

***connector***

Identifies the type of external storage system where the data is located.

Teradata requires the storage location to start with the following for all external storage locations:

- Amazon S3 storage location must begin with `/S3` or `/s3`
- Azure Blob storage location (including Azure Data Lake Storage Gen2 in Blob Interop Mode) must begin with `/AZ` or `/az`.
- Google Cloud Storage location must begin with `/GS` or `/gs`

**storage-account**

Used by Azure. The Azure storage account contains your Azure storage data objects.

**endpoint**

A URL that identifies the system-specific entry point for the external object storage system.

**bucket (Amazon S3, GCS) or container (Azure Blob storage and Azure Data Lake Storage Gen2)**

A container that logically groups stored objects in the external storage system.

**key\_prefix**

Identifies one or more objects in the logical organization of the bucket data. Because it is a key prefix, not an actual directory path, the key prefix may match one or more objects in the external storage. For example, the key prefix '/fabrics/cotton/colors/b/' would match objects: /fabrics/cotton/colors/blue, /fabrics/cotton/colors/brown, and /fabrics/cotton/colors/black. If there were organization levels below those, such as /fabrics/cotton/colors/blue/shirts, the same key prefix would gather those objects too.

**Note:**

Vantage validates only the first file it encounters from the location key prefix.

For example, this LOCATION value might specify all objects on an Amazon cloud storage system for the month of December, 2001:

```
LOCATION('/S3/YOUR-BUCKET.s3.amazonaws.com/csv/US-Crimes/csv-files/2001/Dec/')
```

connector	bucket	endpoint	key_prefixes
S3	YOUR-BUCKET	s3.amazonaws.com	csv/US-Crimes/csv-files/2001/Dec/

This LOCATION could specify an individual storage object (or file), Day1.csv:

```
LOCATION('/S3/YOUR-BUCKET.s3.amazonaws.com/csv/US-Crimes/csv-files/2001/Dec/Day1.csv')
```

connector	bucket	endpoint	key_prefixes
S3	<i>YOUR-BUCKET</i>	s3. amazonaws.com	csv/US-Crimes/csv-files/2001/Dec/ Day11.csv

This LOCATION specifies an entire container in an Azure external object store (Azure Blob storage or Azure Data Lake Storage Gen2). The *container* may contain multiple file objects:

```
LOCATION(' /AZ/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/  
CONTAINER/nos-csv-data')
```

connector	storage-account	endpoint	container	key_prefixes
AZ	<i>YOUR-STORAGE-ACCOUNT</i>	blob.core. windows.net	<i>CONTAINER</i>	nos-csv-data

This is an example of a Google Cloud Storage location:

```
LOCATION('/gs/storage.googleapis.com/YOUR-BUCKET/CSVDATA/  
RIVERS/rivers.csv')
```

connector	endpoint	bucket	key_prefixes
GS	storage.googleapis.com	<i>YOUR-BUCKET</i>	CSVDATA/RIVERS/rivers.csv

## AUTHORIZATION

[Optional] Authorization for accessing external storage.

On any platform, you can specify an authorization object (*[DatabaseName.]AuthorizationObjectName*). You must have the EXECUTE privilege on *AuthorizationObjectName*.

On Amazon S3 and Azure Blob storage and Azure Data Lake Storage Gen2, you can specify either an authorization object or a string in JSON format. The string specifies the USER (*identification*) and PASSWORD (*secret\_key*) for accessing external storage. The following table shows the supported credentials for USER and PASSWORD (used in the CREATE AUTHORIZATION command):

System/Scheme	USER	PASSWORD
AWS	Access Key ID	Access Key Secret
Azure / Shared Key	Storage Account Name	Storage Account Key
Azure Shared Access Signature (SAS)	Storage Account Name	Account SAS Token

System/Scheme	USER	PASSWORD
Google Cloud (S3 interop mode)	Access Key ID	Access Key Secret
Google Cloud (native)	Client Email	Private Key
On-premises object stores	Access Key ID	Access Key Secret
Public access object stores	<empty string> Enclose the empty string in single straight quotes: USER ''	<empty string> Enclose the empty string in single straight quotes: PASSWORD ''

If you use a function mapping to define a wrapper for READ\_NOS, you can specify the authorization in the function mapping. Note that [ INVOKER | DEFINER ] TRUSTED must be used with function mapping.

If you are using AWS IAM credentials, you can omit the AUTHORIZATION clause.

When accessing GCS, Advanced SQL Engine uses either the S3-compatible connector or the native Google connector, depending on the user credentials.

## BUFFERSIZE

Size of the network buffer to allocate when retrieving data from the external storage repository.

The default value is 16 MB, which is the maximum value.

## NOSREAD\_RECORD

[Default] Returns one row for each external record along with its metadata.

Access external records by specifying one of the following:

- Input table and LOCATION and an empty table.
- Input table with a row for each external file.

For an empty single-column input table, do the following:

- Define an input table with a single column, Payload, with the appropriate data type:
  - JSON
  - CSV

This column determines the output Payload column return type.

See [Example: Using READ\\_NOS with NOSREAD\\_RECORD Return Type](#).

- For LOCATION, specify the filepath.

For a multiple-column input table, define an input table with the following columns:

- Location VARCHAR(2048) CHARACTER SET UNICODE

- ObjectVersionID VARCHAR(1024) CHARACTER SET UNICODE
- ObjectTimeStamp TIMESTAMP(6)
- OffsetIntoObject BIGINT
- ObjectLength BIGINT
- Payload JSON

This table can be populated using the output of the NOSREAD\_KEYS return type.

### NOSREAD\_KEYS

Retrieves the list of files from the path specified in the LOCATION USING clause.

A schema definition is not necessary.

Returns:

- Location
- ObjectVersionID
- ObjectTimeStamp
- ObjectLength, size of external file.

See [Example: Using READ\\_NOS with NOSREAD\\_KEYS Return Type](#).

### NOSREAD\_PARQUET\_SCHEMA

Returns information about the Parquet data schema. If you are using the FULLSCAN option, use NOSREAD\_PARQUET\_SCHEMA; otherwise you can use NOSREAD\_SCHEMA to get information about the Parquet schema. For information about the mapping between Parquet data types and Teradata data types, see *Parquet External Files in Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

### NOSREAD\_SCHEMA

Returns the name and data type of each column of the file specified by *external\_file\_path*. Schema format can be JSON, CSV, or Parquet.

### SAMPLE\_PERC

Specifies the percentage of rows to retrieve from the external storage repository when RETURNTYPE is NOSREAD\_RECORD. The valid range of values is from '0.0' to '1.0', where '1.0' represents 100% of the rows.

The default value is 1.0.

### STOREDAS

Specifies the formatting style of the external data.

- PARQUET means the external data is formatted as Parquet. This is a required parameter for Parquet data.

- TEXTFILE means the external data uses a text-based format, such as CSV or JSON.

The default is TEXTFILE.

## FULLSCAN

Determines whether READ\_NOS scans columns of variable length types (CHAR, VARCHAR, BYTE, VARBYTE, JSON, and BSON) to discover the maximum length.

- TRUE means the sizes of variable length data is determined from the Parquet data.

---

### Note:

Choosing this value can impact performance because all variable length data type columns in each Parquet file at the location must be scanned to assess the value having the greatest length.

---

- FALSE means variable length field sizes are assigned the Vantage maximum value for the particular data type.

The default is FALSE.

---

### Note:

FULLSCAN is only used with a RETURNTYPE of NOSREAD\_PARQUET\_SCHEMA.

---

## MANIFEST

Specifies whether the LOCATION value points to a manifest file (a file containing a list of files to read) or object name. The object name can include the full path or a partial path. It must identify a single file containing the manifest.

---

### Note:

The individual entries within the manifest file must show complete paths.

---

Below is an example of a manifest file that contains a list of entries to locations in JSON format.

```
{
  "entries": [
    {"url": "s3://nos-core-us-east-1/UNICODE/JSON/mln-key/
data-10/data-8_9_02-10.json"},
    {"url": "s3://nos-core-us-east-1/UNICODE/JSON/mln-key/
data-10/data-8_9_02-101.json"},
    {"url": "s3://nos-core-us-east-1/UNICODE/JSON/mln-key/data-10/
data-10-01/data-8_9_02-102.json"},
    {"url": "s3://nos-core-us-east-1/UNICODE/JSON/mln-key/data-10/
data-10-01/data-8_9_02-103.json"}
  ]
}
```

```
  ]
}
```

**ROWFORMAT**

Specifies the encoding format of the external row, for example:

```
ROWFORMAT('{"field_delimiter":",",
"record_delimiter":"\n", "character_set":"LATIN"}')
```

Specify ROWFORMAT using JSON format. It can include only the three keys shown above. Key names and values are case-specific, except for the value for "character\_set", which can use any combination of letter cases.

The ROWFORMAT character set specification must be compatible with character set of the Payload column.

Do not specify ROWFORMAT for Parquet format data.

For a JSON column, these are the default values:

Payload Character Set	Default Definition
UNICODE	ROWFORMAT('{"record_delimiter":"\n", "character_set":"UTF8"}')
LATIN	ROWFORMAT('{"record_delimiter":"\n", "character_set":"LATIN"}')

For a CSV column, these are the default values:

Payload Character Set	Default Definition
UNICODE	ROWFORMAT('{"character_set":"UTF8"}') This is the default if you do not specify an input table for READ_NOS.
LATIN	ROWFORMAT('{"character_set":"LATIN"}')

You can specify the following options:

***field\_delimiter***

The default is "," (comma). You can also specify a custom field delimiter, such as tab "\t".



***record\_delimiter***

New line feed character: "\n". A line feed (\n) is the only acceptable record delimiter.

***character\_set***

"UTF8" or "LATIN"

If you do not specify a ROWFORMAT or payload column, Vantage assumes UTF8 Unicode.

**HEADER**

Specifies whether the first row of data in an input CSV file is interpreted as column headings for the subsequent rows of data.

Use this parameter only when a CSV input file is not associated with a separate schema object that defines columns for the CSV data.

The value for HEADER can be 'TRUE' or 'FALSE'. The default is 'TRUE'.

## Setting Up an Object Store for River Flow Data

Many of the examples use a sample river flow data set. USGS Surface-Water Data Sets are provided courtesy of the U.S. Geological Survey.

To run the examples, you can use the river flow data from Teradata-supplied public buckets. Or you can set up a small object store for the data set.

The following instructions explain how to set up the river flow data on your own external object store.

Your external object store must be configured to allow Advanced SQL Engine access.

When you configure external storage, you set the credentials to your external object store. Those credentials are used in SQL commands. The supported credentials for USER and PASSWORD (used in the CREATE AUTHORIZATION command) and for ACCESS\_ID and ACCESS\_KEY (used by READ\_NOS and WRITE\_NOS) correspond to the values shown in the following table:

System/Scheme	USER	PASSWORD
AWS	Access Key ID	Access Key Secret
Azure / Shared Key	Storage Account Name	Storage Account Key
Azure Shared Access Signature (SAS)	Storage Account Name	Account SAS Token
Google Cloud (S3 interop mode)	Access Key ID	Access Key Secret
Google Cloud (native)	Client Email	Private Key

System/Scheme	USER	PASSWORD
On-premises object stores	Access Key ID	Access Key Secret
Public access object stores	<empty string> Enclose the empty string in single straight quotes: USER ''	<empty string> Enclose the empty string in single straight quotes: PASSWORD ''

The following provides further details for setting up credentials on your external object store:

Platform	Notes
Amazon S3 IAM	<p>IAM is an alternative to using an access key and password to secure S3 buckets. To allow Advanced SQL Engine access to S3 buckets that use IAM, your S3 bucket policy must be configured with the following Actions for the role that allows access to the bucket:</p> <ul style="list-style-type: none"> <li>• S3:GetObject</li> <li>• S3:ListBucket</li> <li>• S3:GetBucketLocation</li> </ul> <p>For WRITE_NOS:</p> <ul style="list-style-type: none"> <li>• S3:PutObject</li> </ul> <p><b>Note:</b> Other Actions are also allowed, such as S3:HeadBucket, S3:HeadObject, S3:ListBucket, and so on.</p>
Azure Blob storage and Azure Data Lake Storage Gen2	<p>A user with access key information has full control over the entire account. Alternatively, SAS can be defined on Containers, or on objects within containers, so it provides a more fine-grained authentication approach. NOS uses either type of authentication and does not need to know what type of secret is being supplied.</p> <p><b>Note:</b> Only Account SAS tokens are supported. Service SAS tokens generate errors and are rejected.</p>
Google Cloud Storage	<p>To allow Advanced SQL Engine access, the following permissions are needed:</p> <ul style="list-style-type: none"> <li>• storage.objects.get</li> <li>• storage.objects.list</li> </ul>

See your cloud vendor documentation for instructions on creating an external object store account.

**The following steps may require the assistance of your public cloud administrator.**

1. Create an external object store on a Teradata-supported external object storage platform. Give your external object store a unique name. In the Teradata-supplied examples, the bucket/container is called td-usgs. Because the bucket/container name must be unique, choose a name other than td-usgs.
2. On Amazon, generate an access ID and matching secret key for your bucket or generate an Identity and Access Management (IAM) user credential. On Azure, generate Account SAS tokens (not Service SAS tokens) for your td-usgs container. On Google Cloud Storage, generate an access ID and matching secret key for your bucket.

3. Download the sample data from <https://downloads.teradata.com/> (look for NOS Download Data) to your client/laptop. The ZIP file contains sample river flow data in CSV, JSON, and Parquet data formats.
4. Copy the sample data to your bucket or container, being careful to preserve the data directory structure. For example, use a location similar to the following:
  - Amazon S3: `/S3/YOUR-BUCKET.s3.amazonaws.com/JSONDATA`
  - Azure Blob storage and Azure Data Lake Storage Gen2: `/az/YOUR-STORAGE-ACCOUNT.blob.core.windows.net/td-usgs/CSVDATA/`
  - Google Cloud Storage: `/gs/storage.googleapis.com/YOUR-BUCKET/CSVDATA/`

Note, you can use the Amazon S3 or Azure management consoles or a utility like AWS CLI to copy the data to your external object store. For Google Cloud Storage, you can use the `gsutil` tool to copy the data to your external object store.

5. In the example code replace `td-usgs`, `YOUR-BUCKET`, and `YOUR-STORAGE-ACCOUNT` with the location of your object store.
6. Replace `YOUR-ACCESS-KEY-ID` and `YOUR-SECRET-ACCESS-KEY` with the access values for your external object store.

## Examples: Setting Up Function Mapping for READ\_NOS

### Define a Function Mapping for READ\_NOS

You can use function mapping to hide credentials in an authorization object. After you create the function mapping, you can use the mapped authorization object in queries and avoid exposing your credentials.

The user needs the following privileges to create an authorization object and a function mapping. Contact your administrative user for assistance

- CREATE AUTHORIZATION privilege to create an authorization object
- CREATE FUNCTION privilege to create a function mapping

You can define a function mapping for the `READ_NOS` function with the `EXTERNAL SECURITY` clause to include an authorization. You can also define a function mapping which includes the `ANY IN TABLE` clause for querying files without an extension.

### Example: Creating an Authorization Object to Use with a Function Mapping

Create the authorization object used by the following function mapping examples:

```
CREATE AUTHORIZATION authorization-name
AS DEFINER TRUSTED
```

```
USER 'YOUR-ACCESS-KEY-ID'
PASSWORD 'YOUR-SECRET-ACCESS-KEY';
```

Replace *authorization-name* with the name of your authorization object. Substitute your object store access ID and access key for the variables *YOUR-ACCESS-KEY-ID* and *YOUR-SECRET-ACCESS-KEY*.

For example, to access the Teradata-supplied river flow data set:

```
CREATE AUTHORIZATION nos_usr.DefAuth
AS DEFINER TRUSTED
USER ''
PASSWORD '';
```

## Example: Function Mapping Definition Using EXTERNAL SECURITY Clause for JSON Data Files

This statement defines the function mapping `READ_NOS_json_fm` for the `READ_NOS` table operator to query JSON data files. The function mapping includes the authorization `DefAuth` and a return type of `NOSREAD_RECORD`.

```
CREATE FUNCTION MAPPING READ_NOS_json_fm
FOR READ_NOS
EXTERNAL SECURITY DEFINER TRUSTED DefAuth
USING
  LOCATION('/S3/td-usgs-public.s3.amazonaws.com/JSONDATA'),
  RETURNTYPE('NOSREAD_RECORD') ;
```

## Example: Function Mapping Definition Using ANY IN TABLE for JSON Data Files

This statement defines the function mapping `READ_NOS_json_intable_fm` for the `READ_NOS` table operator. The function mapping includes the `EXTERNAL SECURITY` clause with the authorization `DefAuth`, a return type of `NOSREAD_RECORD`, and the `ANY IN TABLE` clause for querying JSON files regardless of file extension or lack thereof.

```
CREATE FUNCTION MAPPING READ_NOS_json_intable_fm
FOR TD_SYSFNLIB.READ_NOS EXTERNAL SECURITY DEFINER TRUSTED DefAuth
USING
  LOCATION('/s3/td-usgs-public.s3.amazonaws.com/JSONDATA'),
  RETURNTYPE('NOSREAD_RECORD'),
```

```
ANY IN TABLE
;
```

## Examples: Using READ\_NOS

### Example: Create an Authorization Object

Create an authorization object to use with the following examples:

```
CREATE AUTHORIZATION MyAuthObj
USER ''
PASSWORD '';
```

### Example: Using READ\_NOS with NOSREAD\_KEYS Return Type

This is an example of how to use the READ\_NOS table operator with a return type of NOSREAD\_KEYS to list external files at a specified location.

This statement performs a query of a JSON file. The files specified by the LOCATION option include the extensions, .json and .json.gz. If the files have extension other than .json, .json.gz, .csv, or .tsv, you must specify an ON clause where the payload of the input table must match the payload returned by READ\_NOS.

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Perform the READ\_NOS:

```
SELECT location FROM (
LOCATION='/s3/td-usgs-public.s3.amazonaws.com/JSONDATA'
AUTHORIZATION=MyAuthObj
RETURNTYPE='NOSREAD_KEYS'
) AS d;
```

Your result will be similar to the following:

```
Location
-----
/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/09380000/2018/06/27.json
/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/09380000/2018/06/28.json
/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/09380000/2018/06/29.json
/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/09380000/2018/06/30.json
/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/09380000/2018/07/01.json
[...]
```

## Example: Using READ\_NOS with NOSREAD\_RECORD Return Type

This example accesses external data in JSON format. You can also use READ\_NOS to access CSV and Parquet data.

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Perform the READ\_NOS:

```
SELECT TOP 2 * FROM (
  LOCATION='/s3/td-usgs-public.s3.amazonaws.com/JSONDATA/09380000/2018/06/27.json'
  AUTHORIZATION=MyAuthObj
  RETURNTYPE='NOSREAD_RECORD'
) AS d ;
```

### Note:

The NOSREAD\_RECORD return type is the default mode, so it is optional to specify it in the command.

Your result will be similar to the following:

```
      Location /S3/s3.amazonaws.com/td-usgs-public/
JSONDATA/09380000/2018/06/27.json
ObjectVersionId ?
ObjectTimeStamp          ?
OffsetIntoObject          304
  ObjectLength          151
    ExtraField ?
      Payload { "site_no":"09380000", "datetime":"2018-06-27
00:30", "Flow":"15700", "GageHeight":"9.88", "Temp":"10.9",
"Conductance":"668", "Precipitation":"0.00"}
      Location /S3/s3.amazonaws.com/td-usgs-public/
JSONDATA/09380000/2018/06/27.json
ObjectVersionId ?
ObjectTimeStamp          ?
OffsetIntoObject          608
  ObjectLength          151
    ExtraField ?
      Payload { "site_no":"09380000", "datetime":"2018-06-27
01:00", "Flow":"15100", "GageHeight":"9.77", "Temp":"10.8",
"Conductance":"672", "Precipitation":"0.00"}
```

The output is displayed vertically for readability.

## Query and Sample Results

This query uses the READ\_NOS to return selected information for June 28, 2018, where the temperature was over 70 degrees, listed by time:

```
SELECT payload.Flow, payload.datetime, payload.site_no,
payload.Temp, payload.GageHeight
FROM (
  LOCATION='/S3/td-usgs-public.s3.amazonaws.com/JSONDATA/'
  AUTHORIZATION=MyAuthObj
)
AS derived_table
WHERE payload.datetime LIKE '%2018-06-28%'
AND payload.Temp >70
ORDER BY payload.datetime;
```

Your result will be similar to the following:

Payload.Flow	Payload.datetime	Payload.site_no	Payload.Temp	Payload.GageHeight
80.7	2018-06-28 00:00	09423560	71.5	7.39
222	2018-06-28 00:00	09429070	78.2	4.67
0.00	2018-06-28 00:00	09424900	79.5	-0.13
84.4	2018-06-28 00:15	09423560	71.4	7.43
218	2018-06-28 00:15	09429070	78.1	4.70
0.00	2018-06-28 00:15	09424900	78.9	-0.13
84.5	2018-06-28 00:30	09423560	71.4	7.45
238	2018-06-28 00:30	09429070	78.0	4.74
0.00	2018-06-28 00:30	09424900	78.2	-0.13
87.2	2018-06-28 00:45	09423560	71.3	7.49
224	2018-06-28 00:45	09429070	78.0	4.77
0.00	2018-06-28 00:45	09424900	77.8	-0.13
87.7	2018-06-28 01:00	09423560	71.2	7.52
223	2018-06-28 01:00	09429070	77.9	4.86
0.00	2018-06-28 01:00	09424900	76.5	-0.13

## Examples: Using READ\_NOS with NOSREAD\_SCHEMA Return Type

READ\_NOS detects whether the format of the external file is JSON, CSV, or Parquet.

**Example: NOSREAD\_SCHEMA with JSON Schema**

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Run the READ\_NOS command:

```
SELECT * FROM (
  LOCATION='/s3/td-usgs-public.s3.amazonaws.com/JSONDATA/'
  AUTHORIZATION=MyAuthObj
  RETURNSTYPE='NOSREAD_SCHEMA'
) AS d;
```

Your result will be similar to the following:

```
ColPosition NULL
      Name site_no
      DataType int
      Depth          1
NullFound F
      FileType json
      Location /S3/s3.amazonaws.com/td-usgs-public/
JSONDATA/09513780/2018/06/27.json

ColPosition NULL
      Name datetime
      DataType TIMESTAMP(0) FORMAT'Y4-MM-DDBHH:MI'
      Depth          1
NullFound F
      FileType json
      Location /S3/s3.amazonaws.com/td-usgs-public/
JSONDATA/09513780/2018/06/27.json
```

The output is displayed vertically for readability.

**Example: NOSREAD\_SCHEMA with CSV Schema**

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Run the READ\_NOS command:

```
SELECT * FROM (
  LOCATION='/s3/td-usgs-public.s3.amazonaws.com/CSVDATA/'
  AUTHORIZATION=MyAuthObj
```



```

RETURNNTYPE='NOSREAD_SCHEMA'
) AS d;

```

Your result will be similar to the following:

```

ColPosition NULL
      Name site_no
      DataType int
      Depth          1
NullFound F
      FileType json
      Location /S3/s3.amazonaws.com/td-usgs-public/
JSONDATA/09513780/2018/06/27.json

ColPosition NULL
      Name datetime
      DataType TIMESTAMP(0) FORMAT'Y4-MM-DDBHH:MI'
      Depth          1
NullFound F
      FileType json
      Location /S3/s3.amazonaws.com/td-usgs-public/
JSONDATA/09513780/2018/06/27.json

```

The output is displayed vertically for readability.

### Example: NOSREAD\_SCHEMA with Parquet Schema

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Run the READ\_NOS command:

```

SELECT * FROM (
LOCATION='/s3/td-usgs-public.s3.amazonaws.com/PARQUETDATA/'
AUTHORIZATION=MyAuthObj
RETURNNTYPE='NOSREAD_SCHEMA'
) AS d;

```

Your result will be similar to the following:

```

ColPosition      1
      Name GageHeight2
TdatDataType FLOAT
PhysicalType DOUBLE
LogicalType NONE

```

```

Precision          0
  Scale            0
MinLength          0
MaxLength          0
NullFound         1
  FileType parquet
  Location /S3/s3.amazonaws.com/td-usgs-public/PARQUETDATA/

ColPosition        2
  Name Flow
TdatDataType FLOAT
PhysicalType DOUBLE
LogicalType NONE
  Precision          0
  Scale            0
MinLength          0
MaxLength          0
NullFound         1
  FileType parquet
  Location /S3/s3.amazonaws.com/td-usgs-public/PARQUETDATA/

```

The output is displayed vertically for readability.

## Example: Using READ\_NOS to Display Keys in a JSON File without an Extension

This example uses the READ\_NOS table operator with JSON\_KEYS function to display the keys in a JSON file.

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Perform the READ\_NOS:

```

SELECT DISTINCT * FROM JSON_KEYS (
ON (
SELECT payload FROM (
LOCATION='/s3/td-usgs-public.s3.amazonaws.com/DATA/09380000/2018/06/27'
AUTHORIZATION=MyAuthObj
) AS read_nos_query
) AS json_on_keys_query
) AS d ;

```

Result:

```
JSONKeys
-----
"Flow"
"Precipitation"
"Conductance"
"datetime"
"site_no"
"Temp"
"GageHeight"
```

## Example: Using READ\_NOS to Query an External JSON File

This example shows how to use the READ\_NOS table operator to access an external JSON file that includes the extension .JSON.

### File Name Query

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

This query uses READ\_NOS to query the file at the specified location with a return type of NOSREAD\_KEYS. The LOCATION parameter specifies a JSON file stored on Amazon S3.

Perform the READ\_NOS:

```
SELECT location FROM (
  LOCATION='/S3/td-usgs-public.s3.amazonaws.com/JSONDATA/09380000/2018/06/27'
  AUTHORIZATION=MyAuthObj
  RETURNTYPE='NOSREAD_KEYS'
) AS derived_table;
```

The result shows that the file includes the extension, .JSON.

```
Location
-----
/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/09380000/2018/06/27.json
```

### Sample Query and Results

This query uses READ\_NOS to return specific columns searching for entries where the temperature was greater than 11 and the height was more than 10.00, ordered by date and time.

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Perform the READ\_NOS:

```

SELECT payload.Flow, payload.Precipitation, payload.Conductance,
payload.datetime, payload.site_no, payload.Temp, payload.GageHeight
FROM (
LOCATION='/S3/td-usgs-public.s3.amazonaws.com/JSONDATA/09380000/2018/06/27'
AUTHORIZATION=MyAuthObj
) AS derived_table
WHERE payload.Temp > 11.0 AND payload.GageHeight > 10.00
ORDER BY payload.datetime;

```

Your result will be similar to the following:

Payload.Flow	Payload.Precipitation	Payload.Conductance	Payload.datetime
Payload.site_no	Payload.Temp	Payload.GageHeight	
16400	0.00	668	2018-06-27 14:15
09380000	11.8	10.01	
16400	0.00	670	2018-06-27 14:30
09380000	11.8	10.01	
16600	0.00	665	2018-06-27 14:45
09380000	11.9	10.04	
16600	0.00	669	2018-06-27 15:00
09380000	11.9	10.05	
16700	0.00	668	2018-06-27 15:15
09380000	12.0	10.07	
[...]			

## Example: Using READ\_NOS to Query an External CSV File

This example uses the READ\_NOS table operator to query an external CSV file. This query uses READ\_NOS to select specific columns. The LOCATION parameter specifies a CSV file stored on Amazon S3.

If not already done, create the authorization object. See [Example: Create an Authorization Object](#).

Run the READ\_NOS:

```

SELECT Flow, Precipitation, datetime, site_no, GageHeight
FROM (
LOCATION='/S3/td-usgs-public.s3.amazonaws.com/CSVDATA/'
AUTHORIZATION=MyAuthObj
) AS derived_table
WHERE GageHeight > 3.64
ORDER BY datetime;

```

Results:

Flow	Precipitation	datetime	site_no	GageHeight
.00	.00	2018-06-27 00:00	9400568	6.51

.01	.00	2018-06-27 00:00	9394500	4.75
.00	.00	2018-06-27 00:15	9400568	6.51
.01	.00	2018-06-27 00:15	9394500	4.75
.00	.00	2018-06-27 00:30	9400568	6.51
[...]				

## Example: Using READ\_NOS to Query a File in Google Cloud Storage

This SELECT statement uses READ\_NOS to query a file in Google Cloud Storage. Instead of using an authorization object, the authorization credentials are including in the AUTHORIZATION keyword.

```
SELECT payload FROM (
  LOCATION='/gs/storage.googleapis.com/td-usgs/JSONDATA/'
  AUTHORIZATION='{ "ACCESS_ID":"","ACCESS_KEY":"" }'
) AS dt;
```

Depending on the user credentials, Advanced SQL Engine uses either the S3-compatible connector or the native Google connector.

Your result will be similar to the following:

```
{ "site_no":"09380000", "datetime":"2018-06-27 00:00", "Flow":"16200",
  "GageHeight":"9.97", "Temp":"11.0", "Conductance":"669", "Precipitation":"0.00"}
{ "site_no":"09380000", "datetime":"2018-06-28 00:00", "Flow":"15900",
  "GageHeight":"9.91", "Temp":"11.0", "Conductance":"671", "Precipitation":"0.00"}
[...]
```

## SCRIPT

Executes a user-installed script or any LINUX command inside the database.

### Note:

The SCRIPT table operator is unavailable by default to users of Teradata Vantage delivered as-a-service, such as on AWS and Azure. Contact your Teradata account representative to have it enabled.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
SCRIPT ( [ on_clause ] SCRIPT COMMAND ( runtime_literal_command )
  [ RETURNS ( { '*' | 'column_name data_type_specification' } [,...] ) ]
  [ DELIMITER ( 'delimiter_character' ) ]
  [ CHARSET ( { 'UTF-16' | 'LATIN' } ) ]
```

```
[ QUOTECHAR ( 'quote_character' ) ]
[ AUTH ( 'authorization_name' ) ]
[ [AS] alias_name [( column_name [,...] )] ]
)
```

## Syntax Elements

### *on\_clause*

```
ON { table_name | view_name | ( query_expression ) }
   [ AS correlation_name ]
   [ hash_or_partition_by [ order_by | local_order_by ] ]
```

### *hash\_or\_partition\_by*

```
{ { HASH | PARTITION } BY [ order_by | local_order_by ] [,...] |
  PARTITION BY ANY
}
```

### *order\_by*

```
ORDER BY order_by_spec [,...]
```

### *local\_order\_by*

```
ORDER BY local_order_by_spec [,...]
```

### *order\_by\_spec*

```
{ column_name | column_position | sort_expression } [ ASC | DESC ]
[ NULLS { FIRST | LAST } ]
```

### *local\_order\_by\_spec*

```
localOrderByList [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

### ON clause

The SCRIPT function can have only one ON clause (single input). The ON clause can be specified with no options or with:

- HASH BY
- PARTITION BY
- PARTITION BY ANY
- an optional ORDER BY or LOCAL ORDER BY clause

**SCRIPT\_COMMAND**

The script to be executed. The **SCRIPT\_COMMAND** is a required keyword.

***runtime\_literal\_command***

The parameters to **SCRIPT\_COMMAND** can be an executable name followed by the script name and other inputs, or any valid LINUX command.

**RETURNS**

An optional clause.

The names and types of the output columns returned by the script.

\*

Specifies that all columns of the input table should be returned by the **SCRIPT** function.

***column\_name data\_type\_specification***

The information inside the quotation marks. The *data\_type\_specification* is any Teradata data type with any appropriate modifiers, such as size or character type.

**DELIMITER**

An optional clause. The *delimiter\_character* is a tab by default. The delimiter is used to separate the column values in the input and output strings.

**CHARSET**

An optional clause. The default is LATIN. By specifying UTF-16, **SCRIPT** uses the UTF-16 character encoding for all of the data passed to and from the user-installed script, which is recommended when using CHAR or VARCHAR with CHARACTER SET UNICODE.

**QUOTECHAR**

An optional parameter that forces all input and output to the script to be quoted using the specified character.

Using **QUOTECHAR** also enables the database to distinguish between NULL fields and empty VARCHARs. A VARCHAR with length zero is quoted, while NULL fields are not.

If the character is found in the data, it will be escaped by a second quote character.

For example:

He said, "Hey there!"

QUOTECHAR("") with double quotation marks becomes

"He said, ""Hey there!"""

## AUTH

An optional clause that binds an operating system user to the script via authorization objects.

Use the optional AUTH clause with the SCRIPT table operator to specify the fully qualified name of an authorization object in single quotes. You can only specify one AUTH clause.

If the database name is not provided, the name is fully qualified using the current user or database.

The database user executing the script query must have an EXECUTE privilege for the following:

- EXECUTE privilege on the authorization object specified.
- EXECUTE FUNCTION on td\_sysfnlib.script.

A database user may be associated with more than one authorization, but the authorization that is used is the one specified in the AUTH clause.

You can grant the execute privilege on an authorization to a role and then assign a user privileges using that role. Authorization objects belong to database users that create them, but also implicitly to any owner of that database user. The owners can GRANT or REVOKE the EXECUTE privilege on an authorization object on behalf of the database users.

An authorization object called DEFAULT\_AUTH is created for the SYSUIF database. The authorization is used to control which database users can execute scripts under the default operating system user tdatuser. This authorization object is used when *none* is specified in the AUTH clause. Users must have the EXECUTE privilege granted on the SYSUIF.DEFAULT\_AUTH object in order to execute successfully.

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

## AS

An optional introduction to *alias\_name* or *column\_name*.

### *alias\_name*

An alias for the table referenced by *table\_name.column\_name\_alias* is also used to name expressions.

### *table\_name*

The name of the table being referenced.



***view\_name***

The name of the view being referenced.

***query\_expression***

The name of the view being referenced.

***AS correlation\_name***

An optional alias for the ON clause input table.

**HASH BY**

The rows in the ON clause will be redistributed to AMPs based on the hash value of *column\_specification*. The user-installed script file then runs once on each AMP.

**PARTITION BY or PARTITION BY ANY**

In its *column\_specification*, or comma-separated list of column specifications, the group, or groups, over which the function operates.

PARTITION BY is optional. If there is no PARTITION BY clause, then the entire result set, delivered by the FROM clause, constitutes a single group or partition.

PARTITION BY clause is also called the window partition clause.

**ORDER BY**

In its *value\_expression*, the order in which the values in a group, or partition, are sorted.

**LOCAL ORDER BY**

Orders qualified rows on each AMP in preparation to be input to a table function.

**COLUMN SPECIFICATION**

A SQL expression comprising the following two expressions:

- *column\_name*
- *column\_position*

***column\_name, column\_position***

The name of the column or columns or their ordinal position within the join index definition.

***sort\_expression***

The order in which the SQL expressions are sorted.

**ASC**

Results are to be ordered in ascending sort order.

If the sort field is a character string, the system orders it in ascending order according to the definition of the collation sequence for the current session.

The default order is ASC.

## DESC

Results are to be ordered in descending sort order.

If the sort field is a character string, the system orders it in descending order according to the definition of the collation sequence for the current session.

## NULLS FIRST

NULL results are to be listed first.

## NULLS LAST

NULL results are to be listed last.

## Input and Output to the Script File

A script file reads rows from its standard input (stdin) and writes rows to its standard output (stdout). Each input row to the script file is sent as a string with the column values separated by the delimiter specified in the DELIMITER clause. If no DELIMITER clause is specified, the default is a tab. Each input row is separated from the next by a newline character.

The result of the script file must also be a delimited string and the values are converted back to data types specified in the RETURNS clause. Each output row must be separated from the next by a newline character (\n).

Nulls are represented as empty strings. For example, the following example represents the input string for a row with three integer values where the last two are NULL (delimited by colons):

```
1::\n
```

The first value is 1, there is no value between the first and second delimiter, and no value between the second delimiter and the newline character. The input row is equivalent to (1, NULL, NULL).

If the return values from the script file are not in a delimited string format, the script execution fails.

## Data Types

The following Teradata data types are supported by the SCRIPT table operator.

### Teradata Data Type

BYTE
------

Teradata Data Type
BYTEINT
SMALLINT
INTEGER
BIGINT
DECIMAL/NUMERIC
FLOAT/REAL/DOUBLE PRECISION
NUMBER
DATE
TIME
TIME WITH TIME ZONE
TIMESTAMP
TIMESTAMP WITH TIME ZONE
INTERVAL YEAR
INTERVAL YEAR TO MONTH
INTERVAL MONTH
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 SECOND
CHARACTER
VARBYTE
VARCHAR
CLOB

## Setting the Database Search Path

The SEARCHUIFDBPATH option to the SET SESSION sets the database search path for script execution. For details, see *Teradata Vantage™ - SQL Data Definition Language Detailed Topics*, B035-1184.

## Executing LINUX Commands

The text included in the SCRIPT\_COMMAND clause runs in its own shell inside a temporary working directory.

For each database named in the SEARCHUIFDBPATH, a link is added to the working directory that has the same name as that database. You must set SEARCHUIFDBPATH in order to access the user-installed files.

To access files with the SCRIPT table operator, add a relative path using './', the database name, and '/' the file name.

The following is an example of SCRIPT accessing the 'mapper.py' file in the database called 'mydatabase':

```
SELECT * FROM SCRIPT
( ON source
  SCRIPT_COMMAND('python ./mydatabase/mapper.py')
);
```

## Executing Scripts

The SCRIPT table operator runs once per AMP by default. If a PARTITION BY clause is used on the input data, SCRIPT is executed once per partition.

An 'export PATH;' is prepended to whatever you run using the SCRIPT Table Operator. When the bash process spawns a child, the child has the PATH as part of its environment.

The table operator executes scripts under tdatuser by default, so PATH refers to the tdatuser's PATH environment variable. With the AUTH feature, you control which operating system user to execute under the SCRIPT table operator. Generally, the operating system user specified in the SCRIPT table operator's AUTH clause has already exported the PATH environment variable.

### Example: Reading the PATH Variable without Setting It

```
SELECT * from SCRIPT(script_command('echo $PATH;') returns
('PATH varchar(512)'));
```

PATH

```
-----
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.
```

```
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.  
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.
```

## Script Files and Security

The installed script files run in protected mode (outside of the database). The scripts are run as tdatuser.

## Setting Memory Limits

The `setrlimit()` field in `cufconfig` limits the memory for the SCRIPT table operator.

By default, the value is set to 32 MB, but you can increase the value or set it to 0 (no memory limit).

## Troubleshooting

Teradata recommends testing scripts outside of the database on sample data before installing them to prevent errors.

If there is a script error, the script aborts and error information is written to the script log on each node at `/var/opt/teradata/tdtemp/uiflib/scriptlog`.

## Examples

### Example: SCRIPT Table Operator Function

Create a file called `helloworld.py` in the `/root` directory with the following code:

```
#!/usr/bin/python  
print 'hello world!'
```

Install the file in the database:

```
DATABASE mydatabase;  
call SYSUIF.install_file('helloworld',  
                        'helloworld.py', 'cz!/root/helloworld.py');
```

Run the script:

```
SET SESSION SEARCHUIFDBPATH = mydatabase;  
SELECT DISTINCT *  
FROM SCRIPT (  
    SCRIPT_COMMAND('./mydatabase/helloworld.py')
```

```

        RETURNS ('text VARCHAR(30)')
);

```

In the following example, install a script and other files, and then execute the SCRIPT table operator.

```

DATABASE mydb;
SET SESSION SEARCHUIFDBPATH = mydb;
call SYSUIF.install_file('my_analytics',
                        'cz!my_analytics.sh!/tmp/my_analytics.sh');
call SYSUIF.install_file('my_model',
                        'cz!my_model.model!/tmp/my_model.sh');
call SYSUIF.install_file('my_data',
                        'cz!my_data.dat!/tmp/my_data.dat');
Select * from SCRIPT( on data_table
                    SCRIPT_COMMAND('./mydb/my_analytics.sh -mymodel ./mydb/
my_model.model -myadditionaldata ./mydb/my_data.dat')
                    RETURNS('*', 'score varchar(10)');

```

In this example, invoke a Python script file using the SCRIPT table operator. The script mapper.py reads a line of text input (“Old Macdonald Had A Farm”) and splits the line into individual words, emitting a new row for each word.

An example of the Python script:

```

#!/usr/bin/python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
    print '%s\t%s' % (word, 1)

```

To install the script, run the following command:

```
CALL SYSUIF.INSTALL_FILE('mapper', 'mapper.py'
                        'cz!/tmp/mapper.py');
```

The table barrier contains the sentence as one line of text input:

Id int	Name varchar(100)
1	Old Macdonald Had A Farm

To split the sentence into individual words, run the following script:

```
SELECT * FROM SCRIPT
( ON ( SELECT name FROM barrier )
  SCRIPT_COMMAND('./mydb/mapper.py')
    RETURNS ( 'word varchar(10)', 'count_input int' ) ) AS tab;
);
```

The result:

Word	Count_input
Old	1
Macdonald	1
Had	1
A	1
Farm	1

The following example uses some of Python's built-in modules to create a JSON object from a website URL query string. This is useful in converting web URL data to JSON so it can be queried using the JSON data type.

The script requires that Python version 2.6 or higher is installed on the system.

The 'uroltojson.py' Python script:

```
#!/usr/bin/python
import sys
import json
import urlparse
for line in sys.stdin:
    print json.dumps(urlparse.parse_qs(urlparse.urlparse(line.rstrip('\n')).query))
```

SQL to install and run the script on sample data:

```

DATABASE mytestdb;
create table sourcetab(url varchar(10000));
ins sourcetab('https://www.google.com/finance?q=NYSE:TDC');
ins sourcetab('http://www.ebay.com/sch/i.html?
_trksid=p2050601.m570.l1313.TR0.TRC0.H0.Xteradata+merchandise&_nkw=teradata+merc
handise&_sacat=0&_from=R40');
ins sourcetab('https://www.youtube.com/results?
search_query=teradata%20commercial&sm=3');
ins sourcetab('https://www.contrivedexample.com/example?
mylist=1&mylist=2&mylist=...testing');
-- This assumes that urltojson.py is in the current directory.
call sysuif.replace_file('urltojson', 'urltojson.py', 'cz!urltojson.py', 0);
set session searchuifdbpath=mytestdb;
select cast(JSONresult as JSON)
from SCRIPT(
    ON(select url from sourcetab)
    SCRIPT_COMMAND('./mytestdb/urltojson.py')
    RETURNS(' JSONresult VARCHAR(10000)')
);

```

The result:

```

JSONresult
-----
-----
{"q": ["NYSE:TDC"]}
{"mylist": ["1", "2", "...testing"]}
{"_from": ["R40"], "_trksid": ["p2050601.m570.l1313.TR0.TRC0.H0.Xteradata
merchandise"], "_nkw": ["teradata merchandise"], "_sacat": ["0"]}
{"search_query": ["teradata commercial"], "sm": ["3"]}

```

## Example: Use SYSUIF.INSTALL\_FILE to Install Files and then Run the SCRIPT Table Operator

The following example installs a script and other files, and then executes the SCRIPT table operator.

```

DATABASE mydb;
SET SESSION SEARCHUIFDBPATH = mydb;
call SYSUIF.install_file('my_analytics',
                        'cz!my_analytics.sh!/tmp/my_analytics.sh');
call SYSUIF.install_file('my_model',
                        'cz!my_model.model!/tmp/my_model.sh');

```



```
call SYSUIF.install_file('my_data',
                        'cz!my_data.dat!/tmp/my_data.dat');
Select * from SCRIPT( on data_table
                    SCRIPT_COMMAND('./mydb/my_analytics.sh -mymodel ./mydb/
my_model.model -myadditionaldata ./mydb/my_data.dat')
                    RETURNS('*', 'score varchar(10)');
```

## Example: Use the SCRIPT Table Operator to Invoke a Python Script

In this example, invoke a Python script file using the SCRIPT table operator. The script mapper.py reads in a line of text input (“Old Macdonald Had A Farm”) and splits the line into individual words, emitting a new row for each word.

An example of the Python script:

```
#!/usr/bin/python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
    print '%s\t%s' % (word, 1)
```

To install the script, run the following command:

```
CALL SYSUIF.INSTALL_FILE('mapper', 'mapper.py'
                        'cz!/tmp/mapper.py');
```

The table barrier contains the sentence as one line of text input:

Id int	Name varchar(100)
1	Old Macdonald Had A Farm

To split the sentence into individual words, run the following script:

```

SELECT * FROM SCRIPT
( ON ( SELECT name FROM barrier )
SCRIPT_COMMAND('./mydb/mapper.py')
      RETURNS ( 'word varchar(10)', 'count_input int' ) ) AS tab;
);

```

The result:

Word	Count_input
Old	1
Macdonald	1
Had	1
A	1
Farm	1

## Example: Creating JSON Output by Using the SCRIPT Table Operator to Invoke Python Modules

The following example uses Python built-in modules to create a JSON object from a website URL query string. This is useful in converting web URL data to JSON so it can be queried using the JSON data type.

The script requires that Python version 2.6 or higher is installed on the system.

The 'urlojson.py' Python script:

```

#!/usr/bin/python
import sys
import json
import urlparse
for line in sys.stdin:
    print json.dumps(urlparse.parse_qs(urlparse.urlparse(line.rstrip('\n')).query))

```

SQL to install and run the script on sample data:

```

DATABASE mytestdb;
create table sourcetab(url varchar(10000));
ins sourcetab('https://www.google.com/finance?q=NYSE:TDC');
ins sourcetab('http://www.ebay.com/sch/i.html?
_trksid=p2050601.m570.l1313.TR0.TRC0.H0.Xteradata+merchandise&_nkw=teradata+merc
handise&_sacat=0&_from=R40');

```

```

ins sourcetab('https://www.youtube.com/results?
search_query=teradata%20commercial&sm=3');
ins sourcetab('https://www.contrivedexample.com/example?
mylist=1&mylist=2&mylist=...testing');
-- This assumes that urltojson.py is in the current directory.
call sysuif.replace_file('urltojson', 'urltojson.py', 'cz!urltojson.py', 0);
set session searchuifdbpath=mytestdb;
select cast(JSONresult as JSON)
from SCRIPT(
    ON(select url from sourcetab)
    SCRIPT_COMMAND('./mytestdb/urltojson.py')
    RETURNS(' JSONresult VARCHAR(10000)')
);

```

The result:

```

JSONresult
-----
-----
{"q": ["NYSE:TDC"]}
{"mylist": ["1", "2", "...testing"]}
{"_from": ["R40"], "_trksid": ["p2050601.m570.11313.TR0.TRC0.H0.Xteradata
merchandise"], "_nkw": ["teradata merchandise"], "_sacat": ["0"]}
{"search_query": ["teradata commercial"], "sm": ["3"]}

```

## Example: SCRIPT Calls Python for Data Type Conversion

The following example shows the script table operator returning a BYTE and other data types in the return value list:

```

SELECT * FROM SCRIPT(
ON script_syn_res_tab001
SCRIPT_COMMAND('cat')
DELIMITER(',')
RETURNS ('uid int', 'helen BYTE', 'word varchar(10)', 'id int')
) as srpt;

```

In the following Python script, the print command returns HEX numbers for a BYTE (20) return value.

The 'mapper5.py' script:

```
#!/usr/bin/python
import sys
print '040A1b202EC4a256406F7D7Ffffcf3 33 22.4 abc'
```

Save the mapper5.py script in /home/tdatuser or modify the directory location in the following SQL.

Test the script:

```
Select * from SCRIPT(SCRIPT_COMMAND ('/home/tdatuser/mapper5.py' ) returns ('v
byte(20)', 'v2
int', 'v3 float', 'v4 varchar(10)') ) as d1;
```

Result:

v	v2	v3	v4
040A1B202EC4A256406F7D7Ffffcf30000000000	33	2.240000000000000E 001	abc
040A1B202EC4A256406F7D7Ffffcf30000000000	33	2.240000000000000E 001	abc
040A1B202EC4A256406F7D7Ffffcf30000000000	33	2.240000000000000E 001	abc
040A1B202EC4A256406F7D7Ffffcf30000000000	33	2.240000000000000E 001	abc

In the example, the delimiter is a tab. You can specify another character as the delimiter; for example, to specify a comma as the delimiter:

```
#!/usr/bin/python
import sys
print '040A1b202EC4a256406F7D7Ffffcf3,33,22.4,abc'
```

Change the SELECT statement accordingly:

```
SELECT * from SCRIPT(SCRIPT_COMMAND ('/home/tdatuser/
mapper5.py' ) delimiter(',')
returns ('v byte(20)', 'v2 int', 'v3 float', 'v4 varchar(10)') ) as d1;
```

## Related Information

- [Script Installation Procedures](#) for installing scripts.
- *Teradata Vantage™ - JSON Data Type*, B035-1150 about the JSON data type.

# TD\_DBQLParam

Converts BLOB data into a CLOB of JSON format document with 1 to *M* mapping to each row in DBC.DBQLParamTbl.

TD\_DBQLParam is an embedded services system function.

## Result Type

TD\_DBQLParam returns a JSON type document for each data parcel in the supplied BLOB data. If, for example, a request had 1 using row, the resulting number of JSON documents would be 1+1. If a request had *n* using rows, there would be *n* +1 JSON documents.

## ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

## Syntax

```
[TD_SYSFNLIB.] TD_DBQLParam (DBC.DBLParamTbl.QueryID, DBC.DBQLParamTbl.paraminfo)
```

## Syntax Elements

**TD\_SYSFNLIB.**

Name of the database where the function is located.

**DBC.DBLParamTbl.QueryID**

The QueryID column in the following table: DBC.DBQLParamTbl.

**DBC.DBQLParamTbl.paraminfo**

The BLOB column in the following table: DBC.DBQLParamTbl.

## Output

The following help statement provides detailed description of the function argument and output parameters.

```
help function TD_SYSFNLIB.TD_DBQLParam;
```

Column	Data Type	Description
QID	DECIMAL(18,0)	Input parameter QueryID from DBC.DBQLParamTbl.QueryID.
Data	BLOB	Input parameter BLOB data from DBC.DBQLParamTbl.ParamInfo.
QueryID	DECIMAL(18,0)	Output table column #1 (Foreign Key) System wide unique value to join DBQL tables.

Column	Data Type	Description
RowNum	INTEGER	Output table column #2 Row number indicating the sequence of ParamInfo and Data Record JSON documents.
ParamJSON	CLOB	Output table column #3 Parameter and data rows in JSON format.

## Usage Notes

This function is activated as part of DIPSYFNC, the DIP script executed when DIP ALL is executed. But the DIP script does not grant rights to TD\_DBQLParam. DBC will have to be explicitly granted rights. For example:

```
GRANT EXECUTE FUNCTION ON TD_SYSFNLIB.TD_DBQLParam TO DBC WITH GRANT OPTION;
```

For queries that have parameter markers/positional parameters with question marks (?) in the query, variable names are now replaced with proper names only in DBQL's ParamInfo logging to P1, P2 ... Pn. The numbering 1 to *n* stands for these parameter position from left to right as they appear in the query text.

## Example

The following query returns a JSON document for each using row data parcel in:

- The BLOB column paraminfo.
- The associated QueryID that maps into DBC.DBQLLogTbl's QueryID.

```
SELECT JsonTbl.QueryID (FORMAT '--Z(17)9'), JsonTbl.RowNum,
       JsonTbl.ParamJSON from table
       (TD_SYSFNLIB.TD_DBQLParam(DBC.DBQLParamTbl.QueryID,DBC.DBQLParamTbl.paraminfo
       )) as JsonTbl order by JsonTbl.QueryID, JsonTbl.RowNum;
```

Output of a Request with 1 Using Row

QueryID	RowNum	ParamJSON
307192920408671138	1	{"QueryID":"307192920408671138","HostCharSet":"127", "ParamInfo":{"Name": "xABc","Type":"INTEGER","Size":4, "Position":1},{Name": "yBflt","Type":"REAL", "Size":8, "Position":2},{Name": "zCDbl","Type":"REAL", "Size":8, "Position":3}, {"Name": "fxStr", "Type":"CHAR", "Size":20, "Position":4}, {"Name": "varStr", "Type":"VARCHAR", "Size":25, "Position":5}, {"Name": "fxByte", "Type":"BYTE", "Size":4, "Position":6}, {"Name": "vrByte", "Type":"VARBYTE", "Size":25, "Position":7}, {"Name": "nmbr", "Type":"NUMBER", "Size":18, "Position":8}, {"Name": "dcml", "Type":"DECIMAL", "Size":8, "Position":9}, {"Name": "dt", "Type":"DATE", "Size":4, "Position":10}, {"Name": "ts", "Type":"CHAR", "Size":26, "Position":11}, {"Name": "blb", "Type":"BLOB", "Size":60, "Position":12}, {"Name": "clb", "Type":"CLOB", "Size":60, "Position":13}, {"Name": "intrvl", "Type":"CHAR", "Size":5, "Position":14}, {"Name": "tme", "Type":"CHAR", "Size":15, "Position":15}}
307192920408671138	2	{"QueryID":"307192920408671138","Data Record":{"xABc":"1", "yBflt":"+5.7800000000000000E000", "zCDbl":"+9.8670000000000000E-001", "fxStr":null, "varStr": "Test Var String01", "fxByte":"00005AB1", "vrByte":"5ABCfE6789EFBCAB5EF0", "nmbr":"1234.679", "dcml":"54328567.45", "dt":"2013/09/10", "ts":"2013-09-10 10:41:32.

QueryID	RowNum	ParamJSON
		000000", "blb": "BAABBCCDDEEFF123456789AABBCCDDEEFF", "clb": " This is a CLOB column string 01", "intrvl": " 7859", "tme": "10:56:35.000000"}}}

## TD\_DBQLFUL

The FeatureUsage column stores usage information in binary format. You can use the TD\_SYSFNLIB.TD\_DBQLFUL table function to convert the binary data into a JSON document. To display feature use information in JSON format, access the FeatureInfo column of the DBC.QryLogFeatureJSON view.

TD\_DBQLFUL is an embedded services system function.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Result Type

TD\_SYSFNLIB.TD\_DBQLFUL returns a JSON type document as an array of features. For example, if a request used five features, the resulting JSON document has a descriptive name for each of the five features in an array format.

### Syntax

```
[TD_SYSFNLIB.] TD_DBQLFUL ( QueryID
    Decimal(18.0)_type_column,
    VARBYTE_type_column
)
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### QueryID

The QueryID column in the table DBC.DBQLogTbl.

## Output

The following help statement provides detailed description of the function argument and output parameters.

```
help function TD_SYSFNLIB.TD_DBQLFUL;
```

Column	Data Type	Description
QID	DECIMAL(18,0)	Input parameter QueryID from DBC.DBQLogTbl.QueryID.
Data	VARBYTE	Input parameter VARBYTE data from DBC.DBQLogTbl.FeatureUsage.
QueryID	DECIMAL(18,0)	Output table column #1 (Foreign Key) System wide unique value to join DBQL tables.
FeatureUsageJSON	JSON	Output table column #2 feature list in JSON format.

## Usage Notes

This function is activated as part of DIPSYFNC, the DIP script executed when DIP ALL is executed. But the DIP script does not grant rights to TD\_DBQLFUL. DBC will have to be explicitly granted rights. For example:

```
GRANT EXECUTE FUNCTION ON
TD_SYSFNLIB.TD_DBQLFUL TO DBC WITH GRANT OPTION;
```

## Examples

The following query returns a JSON document with feature name array for each FeatureUsage column in DBC.DBQLogTbl.

- The FeatureUsage VARBYTE column.
- The associated QueryID that maps into DBC.DBQLogTbl's QueryID.

```
SELECT JsonTbl.QueryID (FORMAT '--Z(17)9'), JsonTbl.FeatureUsageJSON
FROM
TABLE(TD_SYSFNLIB.TD_DBQLFUL(DBC.DBQLogTbl.QueryID,DBC.DBQLogTbl.FeatureUsage
))
AS JsonTbl ORDER BY JsonTbl.QueryID;
```

This returns a JSON document with a list of features that the request used.

To avoid a complex SQL query, use DBC.QryLogFeatureUseJSON:

```
SELECT QueryID, FeatureJSON
FROM QryLogFeatureUseJSON ;
```

Output of a request with five different features:



QueryID	FeatureUsageJSON
307191834744036453	{"QueryID": "307191834744036453", "FeatureInfo": ["Partition Primary Index", "Multi-Level Partition Primary Index", "User Defined Data Type", "XML Data Type", "Period Data Type"]}

## TD\_UNPIVOT

Transforms table columns into rows.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Required Privileges

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

### Syntax

```
[TD_SYSFNLIB.] TD_UNPIVOT (
  ON { tableName | ( query_expression ) }
  USING VALUE_COLUMNS ( 'value_columns_value' [,...] )
  UNPIVOT_COLUMN ( 'unpivot_column_value' )
  COLUMN_LIST ( 'column_list_value' [,...] )
  [ COLUMN_ALIAS_LIST ( 'column_alias_list_value' [,...] )
    INCLUDE_NULLS ( { 'No' | 'Yes' } )
  ]
)
```

### Syntax Elements

#### TD\_SYSFNLIB.

Name of the database where the function is located.

#### *tableName*

The name of the table with the input data for the function.

#### *query\_expression*

The SELECT statement with input data for the function.

#### VALUE\_COLUMNS

Specifies the destination column or columns for the unpivoted data.

**UNPIVOT\_COLUMN**

Specifies the name of the output column identifying the unpivoted data in each row of the value column of columns.

**COLUMN\_LIST**

Specifies the input column data for the unpivoting operation.

**COLUMN\_ALIAS\_LIST**

Specifies alternate names for the values in the unpivoted column.

**INCLUDE\_NULLS**

Specifies whether or not to include nulls in the transformation. Valid values:

- 'Yes'
- 'No' (default)

## Data Type

The data types of all source columns that are mapped into the same value column must be compatible. For the purposes of TD\_UNPIVOT, two columns are compatible if they are both numeric, both CHAR or VARCHAR, both BYTE or VARBYTE, or both identical.

## Examples

### Example: Twelve Columns Unpivoted to One

In the following example, 12 columns (shown in COLUMN\_LIST) are unpivoted to 1 column with 12 rows.

The input table T in the *query\_expression* (in the SELECT statement below) has the following columns: id, year, jan\_sales, feb\_sales, ..., dec\_sales

```
SELECT * from TD_UNPIVOT(
  ON( select * from T)
  USING
    VALUE_COLUMNS('monthly_sales')
    UNPIVOT_COLUMN('month')
    COLUMN_LIST('jan_sales', 'feb_sales', ..., 'dec_sales')
    COLUMN_ALIAS_LIST('jan', 'feb', ..., 'dec' )
)X;
```

The output columns are the following: id, year, month, monthly\_sales.

One row of input, for example:

```

id
year
jan_sales
feb_sales
... dec_sales
123      2012      100          200          ... 1200

```

generates 12 rows of output:

```

id
year
month
monthly_sales

123 2012   jan    100
123 2012   feb    200
...
123 2012   dec   1200

```

In the following example, there is more than one value column and the `COLUMN_LIST` takes the form: 'a, x, ..', 'b, y, ..', 'c, z, ...', ... where a, x, b, y, c, z, ... are the names of the columns to be unpivoted into the value columns.

Each individual list, for example, 'a, x, ..', defines which columns are mapped to the value columns for one of the rows of output.

The input table T has columns id, year, jan\_sales, jan\_expense, feb\_sales, feb\_expense, ..., dec\_sales, dec\_expense.

```

SELECT * from UNPIVOT(
    ON( select * from T)
    USING
        VALUE_COLUMNS('monthly_sales', 'monthly_expense')
        UNPIVOT_COLUMN('month')
        COLUMN_LIST('jan_sales, jan_expense', 'feb_sales,
feb_expense', ..., 'dec_sales, dec_expense')
        COLUMN_ALIAS_LIST('jan', 'feb', ..., 'dec' )
    )X;

```

The output columns are the following: id, year, month, monthly\_sales, monthly\_expense.

One row of input, for example:

id	year	jan_sales	jan_exp	feb_sales	feb_exp	...	dec_sales	dec_exp
123	2012	100	50	200	100	...	1200	600

generates 12 rows of output:

id	year	month	monthly_sales	monthly_expense
123	2012	jan	100	50
123	2012	feb	200	100
...				
123	2012	dec	1200	600

## Example: More Than One Value Column

In the following example, there is more than one value column and the `COLUMN_LIST` takes the form: 'a, x, ..', 'b, y, ..', 'c, z, ...', ... where a, x, b, y, c, z, ... are the names of the columns to be unpivoted into the value columns.

Each individual list, for example, 'a, x, ..', defines which columns are mapped to the value columns for one of the rows of output.

The input table T has columns id, year, jan\_sales, jan\_expense, feb\_sales, feb\_expense, ..., dec\_sales, dec\_expense.

```
SELECT * from UNPIVOT(
    ON( select * from T)
    USING
        VALUE_COLUMNS('monthly_sales', 'monthly_expense')
        UNPIVOT_COLUMN('month')
        COLUMN_LIST('jan_sales, jan_expense', 'feb_sales,
feb_expense', ..., 'dec_sales, dec_expense')
        COLUMN_ALIAS_LIST('jan', 'feb', ..., 'dec' )
    )X;
```

The output columns are the following: id, year, month, monthly\_sales, monthly\_expense.

One row of input, for example:

id	year	jan_sales	jan_exp	feb_sales	feb_exp	...	dec_sales	dec_exp
123	2012	100	50	200	100	...	1200	600

generates 12 rows of output:

id	year	month	monthly_sales	monthly_expense
123	2012	jan	100	50

123	2012	feb	200	100
...				
123	2012	dec	1200	600

## Usage Notes for Examples

- Columns in table T that are not included in any of the column lists are referred to as "copy columns," and are included in every row of the output.
- Value column names and the UNPIVOT column name must not exist in table T.
- The names in *column\_list\_value* must exist in table T.
- The number of VALUE\_COLUMNS must be equal to the number of columns in each column list. One column list defines which source columns get mapped to the value columns for one row of output.
- Column alias list values are names that are used for the UNPIVOT\_COLUMN to indicate which source column(s) data is in the value columns for that row.

If specified, the number of alias names must be equal to the number of column lists.

If not specified, column alias names default to the names of the columns in the column list, concatenated with an '\_ '.

- The UNPIVOT\_COLUMN is a VARCHAR(1000).
- If INCLUDE\_NULLS = 'yes', there will be exactly  $n$  rows of output for every row of input, where  $n$  is the number of column lists.

If INCLUDE\_NULLS = 'no', then there will be at most  $n$  rows of output for every row of input, where  $n$  is the number of column lists. Any row with NULLS for every value column is eliminated.

## WRITE\_NOS

Use WRITE\_NOS to write data from Vantage to external storage, like Amazon S3, Azure Blob storage, or Google Cloud Storage.

### Required Privileges

You must have the EXECUTE FUNCTION privilege on TD\_SYSFNLIB.WRITE\_NOS.

## WRITE\_NOS Syntax

```
Write_NOS (
  ON { [ database_name.]table_name | (subquery) }
      [ PARTITION BY column_name [ ,... ] ORDER BY column_name [ ,... ] |
        HASH BY column_name [ ,... ] LOCAL ORDER BY column_name [ ,... ] |
        LOCAL ORDER BY column_name [ ,... ] ]
  USING
    LOCATION ('external_storage_path')
    [ AUTHORIZATION ( { [DatabaseName.]AuthorizationObjectName |
      '{"Access_ID":"identification", "Access_Key":"secret_key"}' } ) ]
    STOREDAS ('PARQUET')
    [ NAMING ({ 'DISCRETE' | 'RANGE' }) ]
    [ MANIFESTFILE ('manifest_name') ]
    [ MANIFESTONLY ('TRUE') ]
    [ OVERWRITE ({ 'TRUE' | 'FALSE' }) ]
    [ INCLUDE_ORDERING ({ 'TRUE' | 'FALSE' }) ]
    [ INCLUDE_HASHBY ({ 'TRUE' | 'FALSE' }) ]
    [ MAXOBJECTSIZE ('max_object_size') ]
    [ COMPRESSION ( { 'GZIP' | 'SNAPPY' } ) ]
)
```

### Note:

You must type the bold curly braces shown in `'{"Access_ID":"identification", "Access_Key":"secret_key"}'`. Those particular curly braces are part of the WRITE\_NOS statement.

### Syntax Elements

[ *database\_name*.]*table\_name*

Name of the table from which data will be written to external storage.

**subquery**

Query that returns data to be written to external storage.

**PARTITION BY** *column\_name* [,...] **ORDER BY** *column\_name* [,...]  
**HASH BY** *column\_name* [,...] **LOCAL ORDER BY** *column\_name* [,...]  
**LOCAL ORDER BY** *column\_name* [,...]

These clauses allow you to control how Vantage sorts and divides the rows of *table\_name* or the rows returned by *subquery* among the AMPs before these rows are written to external storage. These clauses affect the external storage object names, which reflect the values each stored object contains according to the NAMING option.

- **PARTITION BY** invokes **WRITE\_NOS** multiple times on each AMP, once for each partition, and writes at least one file to external storage per partition. If you specify **PARTITION BY**, the partitioning columns must be included in the **ORDER BY** list.
- **HASH BY** invokes **WRITE\_NOS** only once per AMP, and writes at least one file to external storage for each AMP.
- Use **LOCAL ORDER BY** alone if any of the columns that would have been in the **PARTITION BY** or **HASH BY** column lists has only a small variety of values, which would hash all the table data to one or just a small number of AMPs, causing data skew. **LOCAL ORDER BY** minimizes data skew by preventing data from being redistributed based on the partitioning or hashing columns. Instead the data is ordered in place on each AMP before being written to external storage.

You can specify up to 10 columns to partition, hash, or local order by.

You can see how many partitions will be created for a table or subquery before you write to external storage by using a query that groups the results by the partitioning columns.

For example:

```
SELECT COUNT(*)
FROM (SELECT * FROM table
      GROUP BY column1,column2)
derived_table_name;
```

The following restrictions on columns used for **ORDER BY** and **LOCAL ORDER BY**:

- Data types are limited to **BYTEINT**, **SMALLINT**, **INTEGER**, **BIGINT**, **DATE**, and **VARCHAR**.
- **VARCHAR** columns can only contain alphanumeric and these special characters:

```
- _ ! * ' ( )
```

- **VARCHAR** columns are limited to 128 characters.

**LOCATION**

A URI identifying the external storage system in the format

```
/connector/endpoint/bucket_or_container/prefix
```

The LOCATION string cannot exceed 2048 characters.

## AUTHORIZATION

[Optional] Authorization for accessing external storage.

On any platform, you can specify an authorization object ([*DatabaseName*.]*AuthorizationObjectName*). You must have the EXECUTE privilege on *AuthorizationObjectName*.

On Amazon S3 and Azure Blob storage and Azure Data Lake Storage Gen2, you can specify either an authorization object or a string in JSON format. The string specifies the USER (*identification*) and PASSWORD (*secret\_key*) for accessing external storage. The following table shows the supported credentials for USER and PASSWORD (used in the CREATE AUTHORIZATION command):

System/Scheme	USER	PASSWORD
AWS	Access Key ID	Access Key Secret
Azure / Shared Key	Storage Account Name	Storage Account Key
Azure Shared Access Signature (SAS)	Storage Account Name	Account SAS Token
Google Cloud (S3 interop mode)	Access Key ID	Access Key Secret
Google Cloud (native)	Client Email	Private Key
On-premises object stores	Access Key ID	Access Key Secret
Public access object stores	<empty string> Enclose the empty string in single straight quotes: USER ''	<empty string> Enclose the empty string in single straight quotes: PASSWORD ''

If you use a function mapping to define a wrapper for READ\_NOS, you can specify the authorization in the function mapping. With function mappings, you can use only [ INVOKER | DEFINER ] TRUSTED, not system-wide authorization.

If an AWS IAM credential provides access, you can omit the AUTHORIZATION clause.

## STOREDAS

Objects created in external storage by WRITE\_NOS are written only in Parquet format.



## NAMING

This determines how the objects containing the rows of data are named in the external storage:

- Discrete naming uses the ordering column values as part of the object names in external storage. For example, if the `PARTITION BY` clause has `ORDER BY dateColumn, intColumn`, the discrete form name of the objects written to external storage would include the values for those columns as part of the object name, which would look similar to this:

```
S3/ceph-s3.teradata.com/
xz186000/2019-03-01/13/object_33_0_1.parquet
```

2019-03-01 is the value for `dateColumn`, the first ordering column, and 13 is the value for the second ordering column, `intColumn`. All rows stored in this external Parquet-formatted object contain those two values.

- Range naming, the default, includes as part of the object name the range of values included in the partition for each ordering column. For example, using the same `ORDER BY` as above the object names would look similar to this:

```
S3/ceph-s3.teradata.com/
xz186000/2019-01-01/2019-03-02/9/10000/object_33_0_1.parquet
```

where 2019-01-01 is the minimum value in that object for the first ordering column, `dateColumn`, 2019-03-02 is the maximum value for the rows stored in this external Parquet-formatted object. Value 9 is the minimum value for the second ordering column, `intColumn`, and 10000 is the maximum value for that column.

## MANIFESTFILE

Specifies the fully qualified path and file name where the manifest file is written. Use the format

```
/connector/end point/bucket_or_container/prefix/manifest_file_name
```

For example:

```
/S3/ceph-s3.teradata.com/xz186000/manifest/manifest.json
```

If you do not include the `MANIFESTFILE` parameter, no manifest file is written.

## MANIFESTONLY

Writes only a manifest file in external storage.

No actual data objects are written to external storage if you use `MANIFESTONLY('TRUE')`.

You must also use the `MANIFESTFILE( 'manifest_name' )` option to create a manifest file in external storage.

Use this option to create a new manifest file in the event that a `WRITE_NOS` operation fails due to a database abort or restart, or when network connectivity issues interrupt and stop a `WRITE_NOS` operation before all data has been written to external storage.

The manifest is created from the table or query result set that is input to `WRITE_NOS`. The input must be a list of storage object names and sizes, with one row per object.

---

**Note:**

The input to `WRITE_NOS` with `MANIFESTONLY` can itself incorporate `READ_NOS`, similar to this, which uses function mappings for `WRITE_NOS` and `READ_NOS`:

```
SELECT * FROM WRITE_NOS_fm
(ON
  (SELECT Location, ObjectLength
   FROM READ_NOS_fm
   (ON
     (SELECT CAST(NULL AS JSON CHARACTER SET UNICODE) )
     USING
       LOCATION( 'your_location' )
       RETURNTYPE( 'NOSREAD_KEYS' )
   ) AS d1 )
  USING
    MANIFESTFILE( 'your_location/orders_manifest.json' )
    MANIFESTONLY( 'TRUE' )
    OVERWRITE( 'TRUE' )
  ) AS d;
```

---

**Note:**

A query like this can be used if a `WRITE_NOS` operation fails before it can create a manifest file. The new manifest file created using `READ_NOS` will reflect all data objects currently in the external storage location, and can aid in determining which data objects resulted from the incomplete `WRITE_NOS` operation. For more information, see *Teradata Vantage™ - Native Object Store Getting Started Guide*, B035-1214.

---

## OVERWRITE

Determines whether an existing manifest file in external storage will be overwritten with a new manifest file that has the same name. If `FALSE`, the default, `WRITE_NOS` returns an error if a manifest file exists in external storage that is named identically to the value of `MANIFESTFILE`.

**Note:**


---

OVERWRITE must be used with MANIFESTONLY( ' TRUE ' ).

---

**INCLUDE\_ORDERING****INCLUDE\_HASHBY**

Determines whether the the ORDER BY and HASH BY columns and their values are written to external storage.

**MAXOBJECTSIZE**

ifies the maximum output object size in megabytes, where *max\_object\_size* is a number between 4 and 16. The default is the value of the DefaultRowGroupSize field in DBS Control. For more information on DBS Control, see *Teradata Vantage™ - Database Utilities*, B035-1102.

**COMPRESSION**

Determines the compression algorithm used to compress the objects written to external storage.

**Note:**


---

For Parquet files the compression occurs inside parts of the parquet file instead of for the entire file, so the file extension on external objects remains .parquet.

---

**Returns**

WRITE\_NOS writes the database data to external storage according to the options passed to it, and returns information about the objects it wrote to external storage. The returned values are a table with the following columns:

**NodeId**

INTEGER value that identifies the SQL Engine node that contains the data written to the external object.

**Ampld**

INTEGER value that identifies the AMP that contains the data written to the external object.

**Sequence**

BIGINT value that uniquely identifies the external object to avoid name conflicts in cases where there might otherwise have been multiple objects with the same path and name.

**ObjectName**

VARCHAR(1024) UNICODE value that is the full object file name in the format:

```
/connector/endpoint/bucket_or_container/prefix/object_file_name
```

- *connector*
- *endpoint*
- *bucket\_or\_container*
- *prefix* depends on the value of the NAMING option, discrete or range naming.
- *object\_file\_name* is in the format:

```
<Location Path>/[ <Discrete Naming or Range  
Naming>]/object_<Node_ID>_<AMP_ID>_<Sequence>.parquet
```

For example:

```
/S3/ceph-s3.mycompany.com/test-bucket/2020-06-01/object_33_0_1.parquet
```

### ObjectSize

BIGINT value that is the external object size in bytes.

### RecordCount

BIGINT value that indicates the number of SQL Engine records in the external object.

## Usage Notes

WRITE\_NOS will not overwrite data objects in external storage. If a WRITE\_NOS operation encounters an object with the exact path and name it is trying to write, it will stop and generate an error.

In this case, the WRITE\_NOS operation will stop, return an error, and will not create a manifest file, even if the MANIFESTFILE option was used. The write operation may have written some data objects to the external storage location, and these must be removed manually before attempting to repeat the same WRITE\_NOS operation. For information on how to determine the data objects that must be removed from external storage before repeating the same WRITE\_NOS operation, see *Teradata Vantage™ - Native Object Store Getting Started Guide*, B035-1214.

## WRITE\_NOS Examples

These examples are for demonstration purposes only. For a fully runnable set of WRITE\_NOS examples, see *Teradata Vantage™ - Native Object Store Getting Started Guide*, B035-1214.

### Write All Data from a Vantage Table to External Storage With Partitioning

The following example writes all the data from a native relational database table to external object storage. The data is partitioned (sorted) by the Vantage Advanced SQL Engine before being written, and the objects containing the data in external storage are named to reflect the sorted data they contain.

```

SELECT NodeId, AmpId, Sequence, ObjectName, ObjectSize, RecordCount
FROM WRITE_NOS_FM (
  ON ( SELECT * FROM RiverFlowPerm )
  PARTITION BY SiteNo ORDER BY SiteNo
  USING
    LOCATION('YOUR-STORAGE-ACCOUNT/RiverFlowPerm_PartitionBy/')
    STOREDAS('PARQUET')
    NAMING('DISCRETE')
    MANIFESTFILE('YOUR-STORAGE-ACCOUNT/
RiverFlowPerm_PartitionBy/flowdata_manifest.json')
    INCLUDE_ORDERING('TRUE')
    MAXOBJECTSIZE('4MB')
    COMPRESSION('GZIP')
) AS d
ORDER BY AmpId;

```

Four rows from an example result set are presented here in a vertical layout (column names on the left) for a more efficient presentation:

NodeId	33
AmpId	0
Sequence	1
ObjectName	YOUR-STORAGE-ACCOUNT/RiverFlowPerm_PartitionBy/09497500/object_33_0_1.parquet
ObjectSize	27682
RecordCount	2944
NodeId	33
AmpId	0
Sequence	1
ObjectName	YOUR-STORAGE-ACCOUNT/RiverFlowPerm_PartitionBy/09513780/object_33_0_1.parquet
ObjectSize	19136
RecordCount	2941
NodeId	33
AmpId	0
Sequence	1
ObjectName	YOUR-STORAGE-ACCOUNT/RiverFlowPerm_PartitionBy/09424900/object_33_0_1.parquet
ObjectSize	23174
RecordCount	2946
NodeId	33

```

      AmpId          0
      Sequence       1
      ObjectName YOUR-STORAGE-ACCOUNT/
      RiverFlowPerm_PartitionBy/flowdata_manifest.json
      ObjectSize     513
      RecordCount    ?

[...]
```

Things to note in this example:

- Each row of the result set describes one object written to external storage. The data objects with names ending in the .parquet extension contain data from Vantage. The object with name ending in the .json extension is the manifest file, which is written after the other data objects.
- Each data object contains data from one partition created according to the PARTITION BY clause. In this case there is one partition for each data collection site, that is, for each SiteNo column value.
- The NAMING( 'DISCRETE' ) parameter causes the object names to include the partition name, in this case the object names include the values for SiteNo. In the object name, *YOUR-STORAGE-ACCOUNT/RiverFlowPerm\_PartitionBy/09497500/object\_33\_0\_1.parquet*, "09497500" is the SiteNo of the data contained by this object.
- Because the query included INCLUDE\_ORDERING( 'TRUE' ), the object data includes data from the ORDER BY column, in this case the site number from which the data in that row was collected.
- The MANIFESTFILE option causes a manifest file to be written to external storage. The manifest file lists all objects that WRITE\_NOS wrote.

### Write All Data from a Vantage Table to External Storage with Hashing

```

SELECT NodeId, AmpId, Sequence, ObjectName, ObjectSize, RecordCount
FROM WRITE_NOS_FM (
  ON ( SELECT * FROM RiverFlowPerm )
  HASH BY SiteNo LOCAL ORDER BY SiteNo
  USING
    LOCATION(' YOUR-STORAGE-ACCOUNT/RiverFlowPerm_HashBy/')
    STOREDAS('PARQUET')
    NAMING('RANGE')
    MANIFESTFILE(' YOUR-STORAGE-ACCOUNT/
RiverFlowPerm_HashBy/flowdata_manifest.json')
    INCLUDE_ORDERING('TRUE')
    INCLUDE_HASHBY('TRUE')
    MAXOBJECTSIZE('4MB')
    COMPRESSION('GZIP')
  ) AS d
ORDER BY AmpId;
```

Four rows from an example result set are presented here in a vertical layout (column names on the left) for a more efficient presentation:

```

      NodeId      33
      AmpId       0
      Sequence    1
      ObjectName  YOUR-STORAGE-ACCOUNT/RiverFlowPerm_HashBy/
09394500/09513780/object_33_0_1.parquet
      ObjectSize   79398
      RecordCount 11778

      NodeId      33
      AmpId       1
      Sequence    1
      ObjectName  YOUR-STORAGE-ACCOUNT/RiverFlowPerm_HashBy/
09380000/09429070/object_33_1_1.parquet
      ObjectSize   84999
      RecordCount  8926

      NodeId      33
      AmpId       2
      Sequence    1
      ObjectName  YOUR-STORAGE-ACCOUNT/RiverFlowPerm_HashBy/
09396100/09400815/object_33_2_1.parquet
      ObjectSize   47659
      RecordCount  6765

      NodeId      33
      AmpId       0
      Sequence    1
      ObjectName  YOUR-STORAGE-ACCOUNT/RiverFlowPerm_HashBy/flowdata_manifest.json
      ObjectSize   624
      RecordCount  ?

[...]
```

Things to note in this example:

- Each row of the result set describes one object written to external storage. The data objects with names ending in the `.parquet` extension contain data from Vantage. The object with name ending in the `.json` extension is the manifest file, which is written after the other data objects.
- Each object contains data grouped according to the `HASH BY` clause. In this example it determines how the database table row data is distributed and grouped to different AMPs using the hash of the

SiteNo column values. In this case there is one partition for the data on each AMP, which may include a combination of data rows from different collection sites.

- The NAMING( ' RANGE ' ) parameter causes the object names to include the upper and lower values in the range of site numbers which data the object contains. In the object name, *YOUR-STORAGE-ACCOUNT/RiverFlowPerm\_HashBy/09394500/09513780/object\_33\_0\_1.parquet* contains data from multiple site numbers between site 09394500 and 9513780, inclusive.
- Because the query included INCLUDE\_ORDERING( ' TRUE ' ) and INCLUDE\_HASHBY( ' TRUE ' ), the object data includes data from the ORDER BY and HASH BY columns. In this case both are the same column, SiteNo, so the data contained in the objects includes the the site number from which the data in that row was collected.
- The MANIFESTFILE option causes a manifest file to be written to external storage. The manifest file lists all objects that WRITE\_NOS wrote.



# User-Defined Functions

## Scalar UDF

Takes input arguments and returns a single value result.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

The RETURNS *data\_type* or RETURNS STYLE clauses are Teradata extensions to the ANSI SQL standard.

### Required Privileges

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke a scalar 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.

### Syntax

```
udf_name ( [ argument [, ...] ] )
```

For UDFs that are defined with parameters of TD\_ANYTYPE data type:

```
( udf_name
  { () RETURNS { data_type | STYLE column_expr } |
    ( argument [, ...] )
    [ RETURNS { data_type | STYLE column_expr } ]
  }
)
```

### Syntax Elements

***udf\_name***

The name of the scalar UDF.

***argument***

A valid SQL expression.

***data\_type***

The desired return type of the TD\_ANYTYPE result parameter.

***column\_expr***

A table or a view column reference that determines the return type of the TD\_ANYTYPE result parameter.

## Restrictions

- Any restrictions that apply to standard SQL scalar functions also apply to scalar UDFs.
- Scalar UDF expressions cannot be used in a partitioning expression of the CREATE TABLE statement.

## UDF Arguments

When Vantage evaluates a UDF expression, it invokes the function with the arguments passed to it. The following rules apply to the arguments in the function call:

- The arguments must be comma-separated expressions in the same order as the parameters declared in the function.
- The number of arguments passed to the UDF must be the same as the number of parameters declared in the function.
- The data types of the arguments must be compatible with the corresponding parameter declarations in the function and follow the precedence rules that apply to compatible types.

## Result Data Type

The result type of a scalar UDF is based on the return type specified in the RETURNS clause of the CREATE FUNCTION statement.

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 in the function call 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 call also includes a TD\_ANYTYPE input argument. 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.

## Default Title

The default title of a scalar UDF appears as:

```
UDF_name
(argument_list
)
```

## Examples

### Example 1

Consider the following table definition and data:

```
CREATE TABLE pRecords (pname CHAR(30),
                        pkey INTEGER);
SELECT * FROM pRecords;
```

The output from the SELECT statement is:

pname	pkey
-----	-----
Tom	6
Bob	5
Jane	4

The following is the SQL definition of a scalar UDF that calculates the factorial of an integer argument:

```
CREATE FUNCTION factorial (i INTEGER)
RETURNS INTEGER
SPECIFIC factorial
LANGUAGE C
NO SQL
PARAMETER STYLE TD_GENERAL
NOT DETERMINISTIC
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'ss!factorial!factorial.c!F!fact'
```

The following query uses the scalar UDF expression to calculate the factorial of the pkey column + 1.

```
SELECT pname, factorial(pkey)+1
FROM pRecords;
```

The output from the SELECT statement is:

pname	(factorial(pkey)+1)
-----	-----
Tom	721
Bob	121
Jane	25

## Example 2

Consider the following table and function definitions:

```
CREATE TABLE T1 (intCol INTEGER,
                  varCharCol VARCHAR(40) CHARACTER SET UNICODE);
CREATE TABLE T2 (intCol INTEGER,
                  decimalCol DECIMAL (10, 6));
CREATE FUNCTION myUDF1 (A INTEGER, B TD_ANYTYPE)
RETURNS TD_ANYTYPE;
CREATE FUNCTION myUDF2 (A TD_ANYTYPE, B TD_ANYTYPE)
RETURNS TD_ANYTYPE;
```

The following invocation of myUDF1 uses the RETURNS *data\_type* clause to specify the UDF return type to be DECIMAL(10,6).

```
SELECT (myUDF1 (T1.intCol, T2.decimalCol) RETURNS DECIMAL(10,6));
```

The following invocation of myUDF2 uses the RETURNS STYLE clause to specify the UDF return type to be the data type of the T1.varCharCol column, which is VARCHAR(40) CHARACTER SET UNICODE.

```
SELECT (myUDF2 (T1.varCharCol, T2.decimalCol)
        RETURNS STYLE T1.varCharCol);
```

## Related Information

- For information on implementing external UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

- For information on CREATE FUNCTION and REPLACE FUNCTION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Administration*, B035-1093.
- For information on EXECUTE FUNCTION and UDTUSAGE privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.
- For information on the TD\_ANYTYPE data type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- To pass an argument that is not compatible with the corresponding parameter type, use CAST to explicitly convert the argument to the proper type. See *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Aggregate UDF

Takes grouped sets of relational data, makes a pass over each group, and returns one result for the group.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

The RETURNS *data\_type* or RETURNS STYLE clauses are Teradata extensions to the ANSI SQL standard.

### Required Privileges

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke an aggregate 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.

### Syntax

```
udf_name ( [ argument [, ...] ] )
```

For UDFs that are defined with parameters of TD\_ANYTYPE data type:

```
( udf_name
  { ( ) RETURNS { data_type | STYLE column_expr } |
    ( argument [, ...] )
    [ RETURNS { data_type | STYLE column_expr } ]
  }
)
```

## Syntax Elements

### *udf\_name*

The name of the aggregate UDF.

### *argument*

A valid SQL expression. See [Usage Notes](#) for rules that apply to aggregate UDF arguments.

### *data\_type*

The desired return type of the TD\_ANYTYPE result parameter.

### *column\_expr*

A table or a view column reference that determines the return type of the TD\_ANYTYPE result parameter.

## Restrictions

- Any restrictions that apply to standard SQL aggregate functions also apply to aggregate UDFs.
- Aggregate UDF expressions cannot appear in a recursive statement of a recursive query. However, a non-recursive seed statement in a recursive query can specify an aggregate UDF.

## Usage Notes

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

The rules that apply to the arguments in an aggregate function call are the same as those that apply to a scalar function call. See [UDF Arguments](#).

The result type of an aggregate UDF is based on the return type specified in the RETURNS clause of the CREATE FUNCTION statement. If the result parameter is of TD\_ANYTYPE data type, see [Result Data Type](#) for information on how the return type is determined.

The default title of an aggregate UDF appears as:

```
UDF_name
(argument_list
)
```

## Example

Consider the following table definition and data:

```
CREATE TABLE Product_Life
(Product_ID INTEGER,
 Product_class VARCHAR(30),
 Hours INTEGER);
SELECT * FROM Product_Life;
```

The output from the SELECT statement is:

Product_ID	Product_class	Hours
100	Bulbs	100
100	Bulbs	200
100	Bulbs	300

The following is the SQL definition of an aggregate UDF that calculates the standard deviation of the input arguments:

```
CREATE FUNCTION STD_DEV (i INTEGER)
RETURNS FLOAT
CLASS AGGREGATE (64)
SPECIFIC std_dev
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL NAME 'ss!stddev!stddev.c!f!STD_DEV'
```

The following query uses the aggregate UDF expression to calculate the standard deviation for the life of a light bulb.

```
SELECT Product_ID, SUM(Hours), STD_DEV(Hours)
FROM Product_Life
WHERE Product_class = 'Bulbs'
GROUP BY Product_ID;
```

The output from the SELECT statement is:

Product_ID	Sum(hours)	std_dev(hours)
100	600	8.16496580927726E 001

## Related Information

- For information on SQL aggregate functions, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- For more information on window aggregate UDFs, see [Window Aggregate UDF](#).
- For more information on implementing aggregate UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information on CREATE FUNCTION AND REPLACE FUNCTION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Administration*, B035-1093.
- For more information on EXECUTE FUNCTION and UDTUSAGE privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.
- For more information on the TD\_ANYTYPE data type, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Window Aggregate UDF

Allows an aggregate UDF with a window specification applied to it to operate on a specified window of rows.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list of an aggregate UDF is empty is a Teradata extension to preserve compatibility with existing applications.

In the presence of an ORDER BY clause and the absence of a ROWS or ROWS BETWEEN clause, ANSI SQL:2011 window aggregate functions use ROWS UNBOUNDED PRECEDING as the default aggregation group, whereas Teradata SQL window aggregate functions use ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

The RESET WHEN clause is a Teradata extension to the ANSI SQL standard.

### Required Privileges

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.

To invoke an aggregate 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.

### Syntax

```
udf_name ( [ argument [, ...] ] ) window
```



**window**

```
OVER (
  [ partition_by_clause ]
  [ order_by_clause ]
  [ rows_clause ]
)
```

***partition\_by\_clause***

```
PARTITION BY column_reference [, ...]
```

***order\_by\_clause***

```
ORDER BY value_expression [, ...] [ ASC | DESC ]
       [ RESET WHEN condition ]
```

***rows\_clause***

```
{ ROWS rows_clause_spec_1 |

  ROWS BETWEEN {
    UNBOUNDED PRECEDING AND rows_clause_spec_2 |
    value PRECEDING AND rows_clause_spec_2 |
    CURRENT ROW AND { UNBOUNDED | value } FOLLOWING
  }
}
```

***rows\_clause\_spec\_1***

```
{ { UNBOUNDED | value } PRECEDING | CURRENT ROW }
```

***rows\_clause\_spec\_2***

```
{ UNBOUNDED FOLLOWING |
  value { PRECEDING | FOLLOWING } |
  CURRENT ROW
}
```

## Syntax Elements

### *udf\_name*

The name of the aggregate UDF on which the window specification is applied.

### *argument*

A valid SQL expression.

## OVER

How values are grouped, ordered, and considered when computing the cumulative, group, or moving function.

Values are grouped according to the PARTITION BY and RESET WHEN clauses, sorted according to the ORDER BY clause, and considered according to the aggregation group within the partition.

## PARTITION BY

In its *column\_reference*, or comma-separated list of column references, the group, or groups, over which the function operates.

PARTITION BY is optional. If there are no PARTITION BY or RESET WHEN clauses, then the entire result set, delivered by the FROM clause, constitutes a single group, or partition.

PARTITION BY clause is also called the window partition clause.

## ORDER BY

In its *value\_expression*, the order in which the values in a group, or partition, are sorted.

## ASC

Ascending sort order. The default is ASC.

## DESC

Descending sort order.

## RESET WHEN

The group or partition, over which the function operates, depending on the evaluation of the specified condition. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition.

RESET WHEN is optional. If there are no RESET WHEN or PARTITION BY clauses, then the entire result set, delivered by the FROM clause, constitutes a single partition.

If RESET WHEN is specified, then the ORDER BY clause must be specified also.

### **condition**

A conditional expression used to determine conditional partitioning. The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify a RESET WHEN clause. In addition, you cannot specify SELECT as a nested subquery within the condition.

The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

For more information, see RESET WHEN in *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145 and the QUALIFY clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

### **ROWS**

The starting point for the aggregation group within the partition. The aggregation group end is the current row.

The aggregation group of a row R is a set of rows, defined relative to R in the ordering of the rows within the partition.

If there is no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

### **ROWS BETWEEN**

The aggregation group start and end, which defines a set of rows relative to the current row in the ordering of the rows within the partition.

The row specified by the group start must precede the row specified by the group end.

If there is no ROWS or ROWS BETWEEN clause, the default aggregation group is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

### **UNBOUNDED PRECEDING**

The entire partition preceding the current row.

### **UNBOUNDED FOLLOWING**

The entire partition following the current row.

### **CURRENT ROW**

The start or end of the aggregation group as the current row.

**value PRECEDING**

The number of rows preceding the current row.

The value for is always a positive integer literal.

The maximum number of rows in an aggregation group is 4096 when *value* PRECEDING appears as the group start or group end.

**value FOLLOWING**

The number of rows following the current row.

The value for value is always a positive integer literal.

The maximum number of rows in an aggregation group is 4096 when value FOLLOWING appears as the group start or group end.

## Arguments to Window Aggregate UDFs

Window aggregate UDFs can take literals, literal expressions, column names (sales, for example), or column expressions (sales + profit) as arguments.

Window aggregates can also take regular aggregates as input parameters to the PARTITION BY and ORDER BY clauses. The RESET WHEN clause can take an aggregate as part of the RESET WHEN condition clause.

The rules that apply to the arguments of the window aggregate UDF are the same as those that apply to aggregate UDF arguments, see [Aggregate UDF](#).

## Supported Window Types for Aggregate UDFs

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> </ul>	Hash partitioning and range partitioning

Window Type	Aggregation Group	Supported Partitioning Strategy
	<ul style="list-style-type: none"> <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>	

Consider the following table definition:

```
CREATE TABLE t (id INTEGER, v INTEGER);
```

The following query specifies a reporting window of rows which the window aggregate UDF MYSUM operates on:

```
SELECT id, v, MYSUM(v) OVER (PARTITION BY id ORDER BY v)
FROM t;
```

The following query specifies a cumulative window of rows which the window aggregate UDF MYSUM operates on:

```
SELECT id, v, MYSUM(v) OVER (PARTITION BY id ORDER BY v
                             ROWS UNBOUNDED PRECEDING)
FROM t;
```

The following query specifies a moving window of rows which the window aggregate UDF MYSUM operates on:

```
SELECT id, v, MYSUM(v) OVER (PARTITION BY id ORDER BY v
                             ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING)
FROM t;
```

## Unsupported Window Types 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>

## 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 use 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.

## Result Type and Format

The result data type of a window aggregate UDF is based on the return type of the aggregate UDF, which is specified in the RETURNS clause of the CREATE FUNCTION statement.

The default format of a window aggregate UDF is the default format for the return type. For information on the default format of data types and an explanation of the formatting characters in the format, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

## Usage Notes

You can apply a window specification to an aggregate UDF. 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.

## Problems With Missing Data

Ensure that data you analyze has no missing data points. Computing a moving function over data with missing points produces unexpected and incorrect results because the computation considers  $n$  physical rows of data rather than  $n$  logical data points.

## Restrictions

- The window feature is *not* supported for aggregate UDFs written in Java.
- Range partitioning for the reporting or cumulative window types is not supported.
- Any restrictions that apply to aggregate UDFs also apply to window aggregate UDFs.
- Any restrictions that apply to the window specification of a standard SQL aggregate function also apply to the window specification of an aggregate UDF.

## Example

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,);
```

The following is the SQL definition of a window aggregate UDF that performs the dense rank operation:

```
REPLACE FUNCTION dense_rank (x INTEGER)
RETURNS INTEGER
CLASS AGGREGATE (1000)
LANGUAGE C
NO SQL
PARAMETER STYLE SQL
DETERMINISTIC
CALLED ON NULL INPUT
EXTERNAL;
```

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. 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

For a C code example of the dense\_rank UDF, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## Related Information

- For information on aggregate UDFs, see [Aggregate UDF](#).
- For more information on ordered analytical functions and the window feature, see *Teradata Vantage™ - SQL Functions, Expressions, and Predicates*, B035-1145.
- For more information on implementing window aggregate UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.
- For more information on CREATE FUNCTION AND REPLACE FUNCTION, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144 and *Teradata Vantage™ - Database Administration*, B035-1093.
- For more information on EXECUTE FUNCTION and UDTUSAGE privileges, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.
- For more information on C code example of the dense\_rank UDF, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## User-Defined Functions

As user-defined function invoked in the FROM clause of a SELECT statement, a Table UDF returns a table to the statement.

See the TABLE option of the FROM clause in *Teradata Vantage™ - SQL Data Manipulation Language*, B035-1146.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

### Required Privileges

You must have EXECUTE FUNCTION privileges on the function or on the database containing the function.



To invoke a table 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.

## Restrictions

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

## Usage Notes

When Vantage evaluates a table UDF expression, it invokes the table function which 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.

A table function can have 128 input parameters. The rules that apply to the arguments in a table function call are the same as those that apply to a scalar function call. See [UDF Arguments](#).

Table UDFs do not have return values. The columns in the result rows that they produce are returned as output parameters.

The output parameters of a table function are defined by the RETURNS TABLE clause of the CREATE FUNCTION statement. The number of output parameters is limited by the maximum number of columns that can be defined for a regular table.

The number and data types of the output parameters can be specified statically in the CREATE FUNCTION statement or dynamically at runtime in the SELECT statement that invokes the table function.

Table Functions and Table Operators cannot execute against fallback data when an AMP is Down. Once the AMP is returned to service the query can complete as normal.

### Example

In this example, the `extract_field` table UDF is used to extract the *customer ID*, *store number*, and *item ID* from the `pending_data` column of the `raw_cust` table.

The `raw_cust` table is defined as:

```
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);
```

The `pending_data` text field is a string of numbers with the 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.

The following shows sample data from the raw\_cust table:

region	pending_data
-----	-----
2	7,2:879,3788,879,4500;08,2:500,9056,390,9004;
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;

The following shows the SQL definition of the extract\_field table UDF:

```
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;
```

The following query extracts and displays the customers and the items they bought from store 25 in region 1.

```

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;

```

The output from the SELECT statement is similar to:

Customer_ID	Item_ID
-----	-----
9005	3789
9004	4907
398	9004
9005	7896
9004	7839

## Related Information

For details on external UDFs, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## UDF Invocation

### UDF Search Precedence

UDFs can be located in any database, so the best practice is to use the fully qualified name of the function, including the containing database, when you invoke any kind of UDF. When a function call is qualified by a database name, Teradata Vantage looks first for the UDF in the specified database.

If you omit the database name, Vantage searches locations to find the UDF using the following order of precedence:

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

# User-Defined Type Expressions/Methods

The following sections describe expressions related to user-defined types (UDTs).

## UDT Expression

Returns a distinct or structured UDT data type.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

### Required Privileges

To use a UDT expression, you must have the UDTTYPE, UDTMETHOD, or UDTUSAGE on the SYSUDTLIB database or the UDTUSAGE privilege on all of the specified UDTs.

### Syntax

```
{ [ database_name. ] table_name ] column_name |
  udf_name ( [ argument_1 [,...] ] ) |
  CAST ( expression AS udt_name ) |
  [ NEW [SYSUDTLIB.] ] constructor_name ( [ argument_2 [,...] ] )
} [ . method_spec [ . ... ] ]
```

#### Note:

You must type the colored or bold period.

#### *method\_spec*

```
method_name ( [ argument_3 [,...] ] )
```

### Syntax Elements

#### *database\_name*

An optional qualifier for the *column\_name*.

**table\_name**

An optional qualifier for the *column\_name*.

**column\_name**

The name of a distinct or structured UDT column.

Constructor methods have the same name as the UDT with which they are associated.

A qualifier for *column\_name*.

**udf\_name**

The name of a distinct or structured UDT data type.

Constructor methods have the same name as the UDT with which they are associated.

**argument**

An argument to the UDF.

An argument to pass to the constructor.

An argument to pass to the method.

Parentheses must appear even though the argument list may be empty.

**CAST**

A CAST expression that converts a source data type to a distinct or structured UDT.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

**expression**

An expression that results in a UDT data type.

**udt\_name**

The name of a distinct or structured UDT data type.

Constructor methods have the same name as the UDT with which they are associated.

**NEW**

An expression that constructs a new instance of a structured type and initializes it using the specified constructor method.

**SYSUDTLIB.**

The database in which the constructor exists.

Vantage only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the expression.

***constructor\_name***

The name of a constructor method associated with a UDT.

Constructor methods have the same name as the UDT with which they are associated.

***method\_name***

The name of an instance method that returns a UDT.

## Usage Notes

You can use UDT expressions as input arguments to UDFs written in C or C++. You cannot use UDT expressions as input arguments to UDFs written in Java.

You can also use UDT expressions as IN and INOUT parameters of stored procedures and external stored procedures written in C or C++. However, you cannot use UDT expressions as IN and INOUT parameters of external stored procedures written in Java.

You can use UDT expressions with most SQL functions and operators, with the exception of ordered analytical functions, provided that a cast definition exists that casts the UDT to a predefined type that is accepted by the function or operator.

### Examples

Consider the following statements that create a distinct UDT named *euro* and a structured UDT named *address* :

```
CREATE TYPE euro
AS DECIMAL(8,2)
FINAL;
CREATE TYPE address
AS (street VARCHAR(20)
,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location* :

```
CREATE TABLE european_sales
(region INTEGER
```

```
,location address
,sales DECIMAL(8,2));
```

## Example: Methods and Functions

The following statement uses the built-in constructor function and mutator methods to return a new instance of the *address* UDT and insert it into the *europaean\_sales* table:

```
INSERT INTO european_sales
VALUES (101, address().street('210 Stanton').zip('76543'), 500);
```

Vantage executes the UDT expression in the following order:

Step	Invocation	Result
1	<i>address()</i> constructor function	Default UDT instance
2	mutator method for <i>street</i>	UDT instance with <i>street</i> attribute set to '210 Stanton'
3	mutator method for <i>zip</i>	UDT instance with <i>zip</i> attribute set to '76543'

The final result of the UDT expression is an instance of the *address* UDT with the *street* attribute set to '210 Stanton' and the *zip* attribute set to '76543'.

## Examples

### Example: Column Name

The following statement creates a table that defines an *address* column named *location*:

```
CREATE TABLE italian_sales
(location address
,sales DECIMAL(8,2));
```

The *location* column reference in the following statement returns an *address* UDT expression.

```
INSERT INTO italian_sales
SELECT location, sales
FROM european_sales
WHERE region = 1151;
```

## Example: CAST

The following statement creates a table that defines a *euro* column named *sales* :

```
CREATE TABLE swiss_sales
(location address
,sales euro);
```

The following statement uses CAST to return a *euro* UDT expression. Using CAST requires a cast definition that converts the DECIMAL(8,2) predefined type to a *euro* type.

```
INSERT INTO swiss_sales
  SELECT location, CAST (sales AS euro)
  FROM european_sales
  WHERE region = 1038;
```

## Example: NEW

The following INSERT statement uses NEW to return an *address* UDT expression and insert it into the *european\_sales* table.

```
INSERT european_sales (1001, NEW address(), 0);
```

## Related Information

FOR more information on ...	SEE ...
NEW	<a href="#">NEW</a> .
method invocation	<a href="#">Method Invocation</a> .
creating a UDT	CREATE TYPE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
creating cast definitions for a UDT	CREATE CAST in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
using UDT expressions in DML statements such as SELECT and INSERT	CREATE TYPE in <i>Teradata Vantage™ - SQL Data Manipulation Language</i> , B035-1146.



## NEW

Constructs a new instance of a structured type and initializes it using the specified constructor method or function.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

### Syntax

```
[ NEW [SYSUDTLIB.] ] constructor_name ( [ argument [,...] ] )
```

### Syntax Elements

#### **SYSUDTLIB.**

Name of the database where the function is located.

#### ***constructor\_name***

The name of a distinct or structured UDT column.

#### ***argument***

An argument to pass to the constructor.

Parentheses must appear even though the argument list may be empty.

## Usage Notes

You can also construct a new instance of a structured type by calling the constructor method or function. For an example, see [Example](#).

To construct a new instance of a dynamic UDT and define the run time composition of the UDT, you must use the NEW VARIANT\_TYPE expression. For more information, see [NEW VARIANT\\_TYPE](#).

## Default Constructor

When a structured UDT is created, Vantage automatically generates a constructor function with an empty argument list that you can use to construct a new instance of the structured UDT and initialize the attributes to NULL.

## Determining Which Constructor is Invoked

Vantage uses the rules in the following table to select a UDT constructor:

IF the NEW expression specifies a constructor with an argument list that is ...	THEN ...
empty	<p>If a constructor method that takes no parameters and has the same name as the UDT:</p> <ul style="list-style-type: none"> <li>Exists in the SYSUDTLIB database, Vantage selects that constructor method.</li> <li>Does not exist in the SYSUDTLIB database, Vantage selects the constructor function that is automatically generated when the structured UDT is created.</li> </ul>
not empty	Vantage selects the constructor method in SYSUDTLIB with a parameter list that matches the arguments passed to the constructor in the NEW expression.

## Examples

### Example

Consider the following statement that creates a structured UDT named *address* :

```
CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location* :

```
CREATE TABLE european_sales
(region INTEGER
 ,location address
 ,sales DECIMAL(8,2));
```

The following statement uses NEW to insert an *address* value into the *european\_sales* table:

```
INSERT european_sales (1001, NEW address(), 0);
```

Vantage selects the default constructor function that was automatically generated for the *address* UDT because the argument list is empty and the *address* UDT was created with no constructor method. The default *address* constructor function initializes the *street* and *zip* attributes to NULL.

The following statement is equivalent to the preceding INSERT statement but calls the constructor function instead of using NEW:

```
INSERT european_sales (1001, address(), 0);
```

To create XML type instances, see [Teradata XML](#).

## Example

To create XML type instances, see [Teradata XML](#).

## Related Information

- For more information on creating constructor methods, and on the constructor function that Vantage automatically generates when the structured type is created, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.
- For more information on constructing a new instance of a dynamic UDT and defining the run time composition of the UDT, see [NEW VARIANT\\_TYPE](#).

## NEW JSON

Constructs a new instance of the JSON data type.

## Related Information

For more information about constructor syntax, see *Teradata Vantage™ - JSON Data Type*, B035-1150.

## NEW VARIANT\_TYPE

Constructs a new instance of a dynamic or VARIANT\_TYPE UDT and defines the run time composition of the UDT.

### ANSI Compliance

This statement is a Teradata extension to the ANSI SQL:2011 standard.

### Syntax

```
NEW VARIANT_TYPE ( variant_type_spec [, ...] )
```

***variant\_type\_spec***

```
{ expression AS alias_name |
  table_name.column_name [ AS alias_name ]
}
```

**Syntax Elements*****expression***

Any valid SQL expression

The following restrictions apply:

- *expression* cannot contain a dynamic UDT expression. Nesting of dynamic UDT expressions is not allowed.
- the first expression (that is, the first attribute of the dynamic UDT) cannot be a LOB, UDT, or LOB-UDT expression.

***alias\_name***

A name representing the expression or column reference which corresponds to an attribute of the dynamic UDT. When provided, *alias\_name* is used as the name of the attribute.

You must provide an alias name for any expression that is not a column reference. You cannot assign the same alias name to more than one attribute of the dynamic UDT. Also, you cannot specify an alias name that is the same as a column name if that column name is already used as an attribute name in the dynamic UDT.

***table\_name***

The name of the table in which the column being referenced is stored.

***column\_name***

The name of the column being referenced. If you do not provide an alias name, the column name is used as the name of the corresponding attribute in the dynamic UDT.

The same column name cannot be used as an attribute name for more than one attribute of the dynamic UDT. If a column has the same name as an alias name, the column name cannot be used as an attribute name.

## Restrictions

- You can use the `NEW VARIANT_TYPE` expression only to construct dynamic UDTs for use as input parameters to UDFs. To construct a new instance of other structured UDTs, use the `NEW` expression. For details, see [NEW](#).
- UDFs support a maximum of 128 parameters. Therefore, you cannot use `NEW VARIANT_TYPE` to construct a dynamic UDT with more than 128 attributes.
- The sum of the maximum sizes for all the attributes of the dynamic UDT must not exceed the maximum permissible column size as configured for the database. Exceeding the maximum column size results in the following SQL error: “ERR\_TEQRWOVRFLW \_T(“Row size or Sort Key size overflow.”)”.

## Usage Notes

You can use the `NEW VARIANT_TYPE` expression to define the runtime composition or internal attributes of a dynamic UDT. Each expression you pass into the `NEW VARIANT_TYPE` constructor corresponds to one attribute of the dynamic UDT. You can assign an alias name to represent each `NEW VARIANT_TYPE` expression parameter. The name of the attribute will be the alias name provided or the column name associated with the column reference if no alias is provided. This is summarized in the following table.

IF...	THEN the attribute name is...
<i>alias_name</i> is provided	<i>alias_name</i>
<i>table_name.column_name</i> is provided, but <i>alias_name</i> is not provided	<i>column_name</i>
an <i>expression</i> is provided that is not a column reference and <i>alias_name</i> is not provided	an error is returned.

Note that you must provide an alias name for all expressions that are not column references. In addition, the attribute names must be unique. Therefore, you must provide unique alias names and/or column references.

The data type of the attribute will be the result data type of the expression. The resultant value of the expression will become the value of the corresponding attribute.

## Examples

### Example

The following `NEW VARIANT_TYPE` expression creates a dynamic UDT with a single attribute named *weight*:

```
NEW VARIANT_TYPE (Table1.a AS weight)
```

In the next example, the `NEW VARIANT_TYPE` expression creates a dynamic UDT with a single attribute named *height*. In this example, no alias name is specified; therefore, the column name is used as the attribute name.

```
NEW VARIANT_TYPE (Table1.height)
```

In the next example, the first attribute is named *height* based on the column name. However, the second attribute is also named *height* based on the specified alias name. This is not allowed since attribute names must be unique; therefore, the system returns the error, "ERRTEQDUPLATTRNAME - "Duplicate attribute names in the attribute list. %VSTR", being returned to the user."

```
NEW VARIANT_TYPE (Table1.height, Table1.a AS height)
```

This example shows a user-defined aggregate function with an input parameter named *parameter\_1* declared as `VARIANT_TYPE` data type. The `SELECT` statement calls the new function using the `NEW VARIANT_TYPE` expression to create a dynamic UDT with two attributes named *a* and *b*.

```
CREATE TYPE INTEGERUDT AS INTEGER FINAL;
CREATE FUNCTION udf_agch002002dynudt (parameter_1  VARIANT_TYPE)
RETURNS INTEGERUDT CLASS AGGREGATE (4) LANGUAGE C  NO SQL
EXTERNAL NAME  'CS!udf_agch002002dynudt!udf_agch002002dynudt.c'
PARAMETER STYLE SQL;
SELECT udf_agch002002dynudt(NEW VARIANT_TYPE (Tbl1.a AS a,
                                              (Tbl1.b + Tbl1.c) AS b))
FROM Tbl1;
```

## Example

This example shows a user-defined aggregate function with an input parameter named *parameter\_1* declared as `VARIANT_TYPE` data type. The `SELECT` statement calls the new function using the `NEW VARIANT_TYPE` expression to create a dynamic UDT with two attributes named *a* and *b*.

```
CREATE TYPE INTEGERUDT AS INTEGER FINAL;
CREATE FUNCTION udf_agch002002dynudt (parameter_1  VARIANT_TYPE)
RETURNS INTEGERUDT CLASS AGGREGATE (4) LANGUAGE C  NO SQL
EXTERNAL NAME  'CS!udf_agch002002dynudt!udf_agch002002dynudt.c'
PARAMETER STYLE SQL;
SELECT udf_agch002002dynudt(NEW VARIANT_TYPE (Tbl1.a AS a,
                                              (Tbl1.b + Tbl1.c) AS b))
FROM Tbl1;
```

## Related Information

- For more information on dynamic UDTs, see `VARIANT_TYPE` in *Teradata Vantage™ - Data Types and Literals*, B035-1143.
- For information on constructing a new instance of a structured UDT that is *not* a dynamic UDT, see [NEW](#).
- For information on writing UDFs which use input parameters of `VARIANT_TYPE` data type, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## NEW XML

Constructs a new instance of the XML data type.

## Related Information

For more information about the functions for the XML data type, see *Teradata Vantage™ - XML Data Type*, B035-1140.

## Method Invocation

Invokes a method associated with a UDT.

### ANSI Compliance

This statement is ANSI SQL:2011 compliant, but includes non-ANSI Teradata extensions.

The requirement that parentheses appear when the argument list is empty is a Teradata extension to preserve compatibility with existing applications.

Additionally, when a statement specifies an ambiguous expression that can be interpreted as a UDF invocation or a method invocation, Teradata gives UDF invocation higher precedence over method invocation. ANSI SQL:2011 gives method invocation higher precedence over UDF invocation.

### Syntax

```
{ [ [ database_name. ] table_name. ] column_name |

  udf_name ( [ argument_1 [, ...] ] ) |

  CAST ( expression AS udt_name ) |

  [ NEW [SYSUDTLIB.] ] constructor_name ( [ argument_2 [, ...] ] )

}.method_name_spec [, ...]
```

***method\_name\_spec***

```
method_name ( [ argument_3 [, ...] ] )
```

**Syntax Elements*****database\_name***

A qualifier for the *column\_name*.

***table\_name***

A qualifier for the *column\_name*.

***column\_name***

The name of a distinct or structured UDT column.

Constructor methods have the same name as the UDT with which they are associated.

***udf\_name***

The name of a distinct or structured UDT data type.

Constructor methods have the same name as the UDT with which they are associated.

***argument***

An argument to the UDF.

An argument to pass to the constructor.

An argument to pass to the method.

Parentheses must appear even though the argument list may be empty.

**CAST**

A CAST expression that converts a source data type to a distinct or structured UDT.

Data type conversions involving UDTs require appropriate cast definitions for the UDTs. To define a cast for a UDT, use the CREATE CAST statement. For more information on CREATE CAST, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

***expression***

An expression that results in a data type that is compatible as the source type of a cast definition for the target UDT.



**NEW**

An expression that constructs a new instance of a structured type and initializes it using the specified constructor method.

**SYSUDTLIB.**

The database in which the constructor exists.

Vantage only searches the SYSUDTLIB database for UDT constructors, regardless of whether the database name appears in the expression.

***constructor\_name***

The name of a constructor method associated with a UDT.

Constructor methods have the same name as the UDT with which they are associated.

***method\_name***

The name of an observer, mutator, or user-defined method (UDM).

You must precede each method name with a period.

## Observer and Mutator Methods

Vantage automatically generates *observer* and *mutator* methods for each attribute of a structured UDT. Observer and mutator methods have the same name as the attribute for which they are generated.

Method	Description	Invocation Example
Observer	Takes no arguments and returns the current value of the attribute.	"Example"
Mutator	Takes one argument and returns a new UDT instance with the specified attribute set to the value of the argument.	"Example: Methods and Functions"

## Usage Notes

When you invoke a UDM on a UDT, Vantage searches the SYSUDTLIB database for a UDM that has the UDT as its first parameter followed by the same number of parameters as the method invocation.

If several UDMs have the same name, Vantage must determine which UDM to invoke. For more information, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## Restrictions

To use any of the following functions as the first argument of a method invocation, you must enclose the function in parentheses:

- DATE
- TIME
- VARGRAPHIC

For example, consider a structured UDT called *datetime\_record* that has a DATE type attribute called *start\_date*. The following statement invokes the *start\_date* mutator method, passing in the result of the DATE function:

```
SELECT datetime_record_column.start_date((DATE)) FROM table1;
```

## Example

Consider the following statement that creates a structured UDT named *address* :

```
CREATE TYPE address
AS (street VARCHAR(20)
   ,zip CHAR(5))
NOT FINAL;
```

The following statement creates a table that defines an *address* column named *location* :

```
CREATE TABLE european_sales
(region INTEGER
 ,location address
 ,sales DECIMAL(8,2));
```

The following statement invokes the *zip* observer method to retrieve the value of each *zip* attribute in the *location* column:

```
SELECT location.zip() FROM european_sales;
```

## Related Information

FOR more information on ...	SEE ...
creating methods	CREATE METHOD in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.

FOR more information on ...	SEE ...
creating UDTs	CREATE TYPE in <i>Teradata Vantage™ - SQL Data Definition Language Syntax and Examples</i> , B035-1144.
UDM programming	<i>Teradata Vantage™ - SQL External Routine Programming</i> , B035-1147.

# Script Installation Procedures

The external stored procedures in the following sections install, replace, re-distribute, and remove user-installed script files.

## Execution Rules

Although the procedures for manipulating scripts are defined as stored procedures, internally they are converted to DDL statements. As a result, all rules that apply to DDL statements apply to these stored procedures. For example, a DDL statement calling the stored procedure must be the last statement in a transaction.

## About Installing and Registering External Language Scripts

External language scripts can be installed, replaced, removed and redistributed to all nodes by using the following external stored procedures.

### SYSUIF.INSTALL\_FILE

Use SYSUIF.INSTALL\_FILE to register the external language script and install the script on all nodes.

The SYSUIF.INSTALL\_FILE external stored procedure has this definition:

```
SYSUIF.INSTALL_FILE(
  IN uif_name VARCHAR(128) NOT CASESPECIFIC CHARACTER SET UNICODE,
  IN uif_filename VARCHAR(256) CASESPECIFIC CHARACTER SET UNICODE,
  IN locspec VARCHAR(1024) CASESPECIFIC CHARACTER SET UNICODE,
);
```

### Syntax Elements

#### *uif\_name*

The SQL name associated with the user-installed file. It cannot have a database name associated with it, as the file is always installed in the current database.

The *uif\_name* should be unique within a database.

The *uif\_name* can be any valid Teradata identifier.

***uif\_filename***

The name of the file and the extension (if any) to register. The *uif\_filename* can be any valid LINUX file name.

Use the *uif\_filename* while executing the file using the SCRIPT table operator.

The max length of *uif\_filename* is 255 characters.

***locspec***

The location of the script file to register.

The first two characters of the *locspec* string specify whether the external language script is on the client or on the database server. If the first two characters are:

- C, then the script is on the client.
- S, then the script 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 delimiter is a single character.

The *file\_type* is 'Z' text file or 'B' binary file.

The remaining characters in the *locspec* string specify the file path.

If the file is on the:

- client, then the file path is a client-interpreted path that specifies the location and name of the file.
- database server, then the file path is a full or relative path specifying the location and name of the file.

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 information on *cufconfig*, see *Teradata Vantage™ - Database Utilities*, B035-1102.

## Example: Calling SYSUIF.INSTALL\_FILE to Install a Python Script

The following statement calls SYSUIF.INSTALL\_FILE to install a Python script and distribute the file to all nodes:

```
CALL SYSUIF.INSTALL_FILE('mapper', 'mapper.py',
'cz!mapper.py!/tmp/mapper.py');
```

## Related Information

For details on administering external stored procedures, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

## About Replacing External Language Scripts

To replace a previously registered external language script, use the external language script SYSUIF.REPLACE\_FILE procedure.

## Before You Begin

Verify that you have the EXECUTE privilege on the REPLACE\_FILE external stored procedure, and the CREATE EXTERNAL PROCEDURE privilege on the current database.

## SYSUIF.REPLACE\_FILE

The SYSUIF.REPLACE\_FILE external stored procedure has the following definition:

```
SYSUIF.REPLACE_FILE (
  IN uif_name VARCHAR(128)NOT CASESPECIFIC CHARACTER SET UNICODE,
  IN uif_filename VARCHAR(256) CASESPECIFIC CHARACTER SET UNICODE,
  IN locspec VARCHAR(1024)CASESPECIFIC CHARACTER SET UNICODE,
  IN override_lock int
);
```

## Syntax Elements

### *uif\_name*

The SQL name associated with the user-installed file. It cannot have a database name associated with it, as the file is always installed in the current database.

The *uif\_name* should be unique within a database.

The *uif\_name* can be any valid Teradata identifier.

### *uif\_filename*

The name of file with the extension (if any). This can be any valid LINUX file name.

***locspec***

The location of the replacement file. The format is identical to the `SYSUIF.INSTALL_FILE locspec` parameter.

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 external language script is on the client.
- SJ, then the external language script 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 external language script is on the:

- client, then the file path is a client-interpreted path that specifies the location and name of the file.
- database server, then the file path can be a full or relative path that specifies the location and name of the file.

If the file path is relative, the full path to the 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 information on *cufconfig*, see *Teradata Vantage™ - Database Utilities*.

If the replacement file does not exist in the current database, the `REPLACE_FILE` creates and registers a new file with the database.

The new file is updated and distributed to all nodes. If a node is down, the pending file copy is recorded and copied as soon as the node comes back up.

***override\_lock***

An integer value for the override lock.

If a script is being executed in the database where the file is being replaced, then all files in that database are locked with read-only access. `REPLACE_FILE` must wait until the script is executed and permission is granted to write the file. If *override\_lock* is set to 1, then the file is replaced even if a script is being executed.

The value of *override\_lock* is 1 or 0.

## Example: Replacing an Existing File Mapper

The following statement replaces the existing file mapper with the new file mapper.py:

```
CALL SYSUIF.REPLACE_FILE('mapper', 'mapper.py',
'cz!mapper.py!/tmp/mapper.py',0);
```

## About Redistributing External Language Scripts

You might need to redistribute an installed file to all nodes of a database system during copying, migration, or system restore operations.

To redistribute a previously registered file to all nodes of a database system, use the REDISTRIBUTE\_FILE external stored procedure. The file to be redistributed must have been registered using the INSTALL\_FILE external stored procedure.

## Before You Begin

You must have the EXECUTE privilege on the REDISTRIBUTE\_FILE procedure and the CREATE EXTERNAL PROCEDURE privilege on the current database.

## SYSUIF.REDISTRIBUTE\_FILE

The SYSUIF.REDISTRIBUTE\_FILE system external stored procedure has the following definition:

```
SYSUIF.REDISTRIBUTE_FILE(
  IN uif_name VARCHAR(128)NOT CASESPECIFIC CHARACTER SET UNICODE);
```

## Syntax Elements

### *uif\_name*

The SQL name associated with the user-installed file. It cannot have a database name associated with it, as the file is always installed in the current database.

The *uif\_name* should be unique within a database.

The *uif\_name* can be any valid Teradata identifier.

If the script file does not exist in the current database, the REDISTRIBUTE\_FILE operation fails.

If the script file is found, the installed file is retrieved from its internal table location and redistributed to all nodes of the system. If a node is down, the file is copied over when the node comes back up.



## Example

Consider a file that was registered with the JXSP database using the following statements:

```
CALL SYSUIF.REDISTRIBUTE_FILE('mapper');
```

## Displaying the User-Installed File

To display the user-installed file and the DDL that created it, use a SHOW FILE request.

## Related Information

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

## About Removing External Language Scripts

To remove an external language script that was previously registered using SYSUIF.INSTALL\_FILE, run SYSUIF.REMOVE\_FILE.

## Before You Begin

Complete the following steps before you remove a registered external language script:

1. Verify that you have the EXECUTE privilege on the REMOVE\_FILE external stored procedure.
2. Check if the default database is where the script file was registered. If not, use the DATABASE statement to change the default database.
3. Verify that you have the DROP PROCEDURE or DROP FUNCTION privilege on either:
  - The default database
  - The file that you are removing from the default database.

## SYSUIF.REMOVE\_FILE

The SYSUIF.REMOVE\_FILE external stored procedure has this definition:

```
SYSUIF.REMOVE_FILE(  
  IN uif_name VARCHAR(128) NOT CASESPECIFIC CHARACTER SET UNICODE,  
  IN override_lock int);
```

## Syntax Elements

### *uif\_name*

The SQL name associated with the user-installed file. It cannot have a database name associated with it, as the file is always installed in the current database.

The *uif\_name* should be unique within a database.

The *uif\_name* can be any valid Teradata identifier.

### *override\_lock*

An integer value for the override lock.

If set to 1, the system does not check for the file being used before removing it.

## Example

The following statement removes the registered file:

```
CALL SYSUIF.REMOVE_FILE('mapper',1);
```

# Notation Conventions

## How to Read Syntax

This document uses the following syntax conventions.

Syntax Convention	Meaning
KEYWORD	Keyword. Spell exactly as shown. Many environments are case-insensitive. Syntax shows keywords in uppercase unless operating system restrictions require them to be lowercase or mixed-case.
<i>variable</i>	Variable. Replace with actual value.
<i>number</i>	String of one or more digits. Do not use commas in numbers with more than three digits. Example: 10045
[ x ]	x is optional.
[ x   y ]	You can specify x, y, or nothing.
{ x   y }	You must specify either x or y.
x [...]	You can repeat x, separating occurrences with spaces. Example: x x x See note after table.
x [, ...]	You can repeat x, separating occurrences with commas. Example: x, x, x See note after table.
x [delimiter...]	You can repeat x, separating occurrences with specified delimiter. Examples: <ul style="list-style-type: none"> <li>If <i>delimiter</i> is semicolon: x; x; x</li> <li>If <i>delimiter</i> is { ,   OR }, you can do either of the following: <ul style="list-style-type: none"> <li>x, x, x</li> <li>x OR x OR x</li> </ul> </li> </ul> See note after table.

**Note:**

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat x. Do not repeat KEYWORD.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[ x, [...] ] y
```

- You can omit x: y
- You can specify x once: x, y
- You can repeat x and the delimiter: x, x, x, y

## Character Shorthand Notation Used in This Document

This document uses the Unicode naming convention for characters. For example, the lowercase character 'a' is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the document, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

### Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a-z A-Z 0-9	Any	Any single byte Latin letter or digit.
<u>a-z</u> <u>A-Z</u> <u>0-9</u>	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by <code>ss3</code> .
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by <code>ss2</code> , forming an individual multibyte character.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

## Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

## 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