

Teradata Vantage™ - DATASET Data Type

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2016 - 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.

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

Chapter 1: Introduction to the Teradata DATASET Data Type	5
Changes and Additions	5
Chapter 2: The DATASET Data Type	6
Teradata Support for the DATASET Data Type	6
Standards Compliance	8
DATASET Data Type Specifications	8
Privileges Required for Creating and Using Schemas	10
DATASET Data Type Syntax	11
Data Definition Language Statements	13
Character Set Handling	13
About the DATASET Type CreateDATASET Function	13
About DATASET Type Transform	13
About DATASET Type Cast	15
About DATASET Type Ordering	15
About DATASET Type Usage	16
FNC Library Routines That Support the DATASET Type	18
Restrictions for the DATASET Type	20
Chapter 3: Operations on the DATASET Data Type	21
Creating and Altering Tables to Store DATASET Data	21
Using Algorithmic Compression on DATASET Columns	23
Accessing DATASET Data Using Dot Notation	23
Modifying DATASET Columns	30
DATASET Methods, Functions, and Table Operators	31
Chapter 4: DATASET Methods	33
AvroProject	33
AvroProjectToJSON	35
DATASET Constructor	36
ExtractValue	38
getRawData	40
getRawDataLob	41
getRawDataSize	42
getSchema	42
getSchemaSize	44
Header	45
numRecords	45
toAvro	46

toJSON	46
Validate	48
Chapter 5: DATASET Functions and Operators	50
AVRO_CHECK	50
AvroContainerSplit	51
CreateDATASET	53
CSVSplit	59
DATASET_KEYS	61
DATASET_TABLE	64
DataSize	72
SchemaEqual	74
SchemaMatch	76
Chapter 6: CSV Conversion	80
CSV Schema	80
CSV_TO_AVRO	82
CSV_TO_JSON	86
Chapter 7: DATASET Publishing	92
About Publishing	92
DATASET_PUBLISH	92
Chapter 8: Avro Object Container Files	104
About Importing and Exporting	104
Importing From an Avro Object Container File	104
Exporting to an Avro Object Container File	106
Appendix A: Notation Conventions	108
Appendix B: External Representations for the DATASET Type	111
Appendix C: Additional Information	115

Introduction to the Teradata DATASET Data Type

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.

Advanced SQL Engine is a core capability of Teradata Vantage, based on our best-in-class Teradata Database. Advanced SQL refers to the ability to run advanced analytic functions beyond that of standard SQL.

For information on data type mapping between Advanced SQL Engine and ML Engine, see *Teradata Vantage™ User Guide*, B700-4002.

Teradata Vantage™ - DATASET Data Type describes support for DATASET data. DATASET is a complex data type (CDT) representing self-describing files that are interpreted based on a schema. Teradata support includes the DATASET data type and the functions and methods available for processing, shredding, and publishing DATASET data.

Changes and Additions

Date	Description
July 2021	Minor edits.

The DATASET Data Type

DATASET is a complex data type provided by Vantage, and is used the same way as other complex data types.

- The DATASET data type exists in the database, so it cannot be created, dropped, or altered by the user.
- DATASET content is stored in the database in an optimized format depending on the size of the data.

Teradata Support for the DATASET Data Type

The Teradata DATASET data type is a complex data type (CDT) representing self-describing files that are interpreted based on a schema. The feature provides the following functionality to support the storage and processing of DATASET data in the database.

Function	Description
Storage and processing	<ul style="list-style-type: none"> • Store variable data formats. Avro and Comma Separated Value (CSV) formats are supported. • Specify the CDT variable maximum length or in-row length. • Define schemas at the column-level or instance-level for any of the built-in storage formats of the DATASET type. Column-level schemas are binding for all instances of the data type loaded into that particular column, while instance-level schemas may vary from instance to instance.
Methods, functions, and stored procedures	Operate on the DATASET type, in any storage format and with any schema.
Shredding	Extract values from DATASET documents and store the extracted data in a relational format.
Publishing	Publish data stored in relational tables and compose a DATASET type with any storage format and any schema.
Analytics	<ul style="list-style-type: none"> • Apply advanced analytics to DATASET data. • Collect statistics on extracted portions of the DATASET type.
SQL	Use standard SQL to query DATASET data.

The feature also provides enhanced dot notation to allow easy access to data. Dot notation includes the following syntax for both DATASET and JSON:

- Recursive descent operator (..)
- Wildcards (*), both in reference to named and indexed items
- Name/index lists ([a,b,c] or [0,3,5])
- Name/index slices ([c] or [5])

Client Support for the DATASET Data Type

Client Product	DATASET Support Provided
CLI	Full native DBS support.
ODBC	<ul style="list-style-type: none"> The ODBC specification does not have a unique data type code for DATASET. Therefore, the ODBC driver maps the DATASET data type to SQL_LONGVARCHAR or SQL_WLONGVARCHAR, which are the ODBC CLOB data types. The metadata differentiates between a Teradata CLOB data type mapped to SQL_LONGVARCHAR and a Teradata DATASET data type mapped to SQL_LONGVARCHAR. The ODBC driver supports LOB Input, Output and InputOutput parameters and can load DATASET data. Catalog (Data Dictionary) functions also support DATASET.
JDBC	<ul style="list-style-type: none"> Teradata JDBC Driver 15.10.00.23 and later support the DATASET data type. The Teradata JDBC Driver offers functionality for an application to use the PreparedStatement or CallableStatement setObject method to bind a Struct value to a question-mark parameter marker as a DATASET data type. An application can also insert VARBYTE or BLOB values into DATASET destination columns. When an application uses the Teradata-specific functionality of specifying a DATASET value as a Struct value, the Struct value must contain one of the following attributes: Byte Array, InputStream, BLOB, or null. If the Struct contains an InputStream attribute, the Struct must also contain a second attribute that is an Integer type specifying the number of bytes in the stream. DATASET values are retrieved from the database as BLOB values. An application can use result set metadata or parameter metadata to distinguish a BLOB value from a DATASET value.
.NET Data Provider	<ul style="list-style-type: none"> The DATASET data type is externalized as a BLOB or VARBYTE. Applications can use TdBlob or TdDataReader.GetBytes to retrieve a DATASET value. Applications can send a DATASET value as BYTE[] to the database. Schema Collections (Data Dictionary) also support the DATASET data type.
Teradata Parallel Transporter (TPT)	DATASET columns are similar to CLOB columns and subject to the same limitations. DATASET columns cannot exceed 16 MB (16,776,192 LATIN characters or 8,388,096 UNICODE characters). When loading or exporting DATASET columns, TPT users should specify CLOB or VARCHAR in the TPT schema definition.
BTEQ	The DATASET keyword cannot be used in the USING data statement; therefore, DATASET values must be referred to as either BLOB or VARBYTE.
Standalone Utilities	No support.

Terminology

Data content and formats constantly evolve, creating different file types. Some file types are proprietary or specific to particular industries or applications, while others have a more general use.

Some applications use particular self-describing file formats. There is no one best solution; using different data types allows for more flexibility. Avro and CSV formats are examples of self-describing data; given the

schema, a set of bytes are interpreted as a set of items described in that schema. The schema is provided with the data, which makes the data self-describing so various applications can understand it.

Regardless of format, purpose, content, or frequency of use, a large amount of self-describing data is analyzed. The database stores and operates on data in its native format using dot notation.

Standards Compliance

The conversion routines are compliant with the standards for CSV data structure used in the conversion routines provided for CSV format data, defined by IETF RFC 4180. The standard is available at <https://tools.ietf.org/html/rfc4180>.

Apache provides specifications for the Avro format. DATASET supports the Apache Avro 1.7.7 specification.

DATASET adds the following non-reserved keywords:

- DOT
- NOTATION
- LIST
- AVRO
- CREATEDATASET
- CSV
- DATASET
- SNAPPY_COMPRESS
- SNAPPY_DECOMPRESS

DATASET Data Type Specifications

Each DATASET data type must be accompanied by the following specifications:

- Maximum length
- In-line length
- Storage format
- Variable schema formats

Maximum and In-Line Length of a DATASET Instance

DATASET uses complex data type enhancements for varying maximum and in-line lengths. The values represent the schema and data byte size for each instance of the DATASET data type.

There is additional space for other information within the instances, but specified values apply to just the schema and data.

Minimum or Maximum Length Type	Length Size
Maximum LOB length	2 GB

Minimum or Maximum Length Type	Length Size
Maximum row size	64 K
Default maximum length (if length is not provided)	2 GB
Default inline length (if length is not provided)	10 K
Minimum length	100 bytes
Maximum size of a schema that is not column-based	16 MB
Maximum size of a binary-encoded Avro value	16 MB
Maximum size of a CSV value	16 MB

Storage Formats

Variable Storage Formats

Each DATASET use must specify a storage format. The STORAGE FORMAT syntax was extended to support the DATASET data type. Vantage provides built-in storage formats for the DATASET data type.

The storage format specification does not necessarily affect the data format on disk, but associates particular data with a specific well-known format.

Built-In Storage Formats

Vantage provides the Avro and CSV storage formats for the DATASET data type, which are based on the Apache Avro and CSV specifications. Each instance contains a schema conforming to the specification. The schema is always optional for the CSV storage format. The schema is interpreted on a per-instance basis, or at the column level.

Storage Format Terminology

Term	Description
Schema	For storage format AVRO, the schema is a JSON document describing the binary-encoded Avro value format. Specified in JSON text, in UTF-8 encoded characters using a VARBYTE or BLOB data type. For CSV, the JSON document describes the extended CSV options such as a field or record delimiter, and column names or header information. It can be specified in any supported JSON format. It is stored in the same character set as the CSV data type for instance-level DATASET values and as UNICODE text, encoded in UTF-8, if stored in the Data Dictionary for column-level DATASET values.
Binary-encoded Avro Value	The actual Avro data, encoded according to the scheme described by the schema.
CSV Value	The CSV value in the Latin or Unicode character set.
JSON-encoded Value	JSON-text representation of the data, as described by the schema.

Term	Description
Transform format OR Cast format	For storage format AVRO, this is a null-terminated, UTF-8 encoded schema followed immediately by a binary-encoded value. For CSV, the transform and cast format uses the original CSV value. If a schema is specified for a CSV value, it is not included in the cast or transform.

Variable Schema Formats

The DATASET data type may be associated with any known schema. You can provide a schema based on frequently used structures, which are stored in the Data Dictionary. All DATASET functionality is available to storage formats using any schema that conforms to specifications for that storage format. After it is registered, any schema can be referenced using the WITH SCHEMA <name> clause.

Note:

You cannot specify the WITH SCHEMA option when creating a volatile table.

No built-in schemas are provided for use with DATASET, in any of its storage formats.

Privileges Required for Creating and Using Schemas

The DATASET data type introduces DDL statements, and added three privileges to the current data control language (DCL).

CREATE DATASET SCHEMA Privilege

CREATE DATASET SCHEMA is a privilege that allows users permission to create a schema in SYSUDTLIB. It is granted at the database level only. The privilege is automatically given to the database DBC with a grant option, and must be explicitly granted with or without grant options by DBC to any other users or databases created, or without grant options to any role created.

The following list provides a summary of this privilege:

- Privilege: CREATE DATASET SCHEMA
- Abbreviation in System Views: C1
- Automatically Granted
 - Creators: No
 - Created User or Database: No
- Explicitly Granted
 - Creators: Yes
 - Created User or Database: Yes
 - Privilege Category: Dataset Schema

DROP DATASET SCHEMA Privilege

The DROP DATASET SCHEMA privilege gives users permission to drop a schema from SYSUDTLIB, and is granted at the database- or individual schema-level. It is automatically given to the database DBC with a grant option, and must be explicitly granted with or without grant options by DBC to any users or databases created or without grant options to any role created. The schema creator is automatically granted this privilege with grant option on the created schema. Note that possession of this privilege does not guarantee the ability to drop a schema.

The following list provides a summary of this privilege:

- Privilege: DROP DATASET SCHEMA
- Abbreviation in System Views: D1
- Automatically Granted
 - Creators: Yes
 - Created User or Database: No
- Explicitly Granted
 - Creators: Yes
 - Created User or Database: Yes
 - Privilege Category: Dataset Schema

WITH DATASET SCHEMA Privilege

The WITH DATASET SCHEMA privilege gives users permission to associate a created schema with a table column, and can be granted at the database- or individual schema-level. It is automatically given to the database DBC with a grant option, and must be explicitly granted with or without grant options by DBC to any users or databases created or without grant options to any role created. The schema creator is automatically granted this privilege with grant option on the created schema.

The following list provides a summary of this privilege:

- Privilege: WITH DATASET SCHEMA
- Abbreviation in System Views: W1
- Automatically Granted
 - Creators: Yes
 - Created User or Database: No
- Explicitly Granted
 - Creators: Yes
 - Created User or Database: Yes
 - Privilege Category: Dataset Schema

For more information, see *Teradata Vantage™ - SQL Data Control Language*, B035-1149.

DATASET Data Type Syntax

The following shows the syntax when you use a DATASET data type in a data type declaration phrase. For example, this syntax is used when defining a table column to be DATASET type.

```
DATASET
[ ( maxlength ) ]
[ INLINE LENGTH integer ]
STORAGE FORMAT [ AVRO | CSV [ CHARACTER SET { LATIN | UNICODE } ] ]
[ attributes ] [...]
```

Syntax Elements

maxlength

[Optional] A maximum length may be specified, in terms of bytes, subject to the absolute maximum of 2 GB which is chosen based on the maximum size of a LOB in Teradata. If not specified, the default maximum length is the absolute maximum.

INLINE LENGTH *integer*

[Optional] An inline length may be specified, in terms of bytes, subject to the absolute maximum of 64,000. If not specified, the default is 10,000.

STORAGE FORMAT

The built-in storage format. The supported types are AVRO and CSV.

CHARACTER SET

[Optional] The optional character set for the CSV storage format. The supported character sets are UNICODE or LATIN. The default character set used is the current server character set of the session.

attributes

The following data type attributes are supported for the DATASET type.

- NULL and NOT NULL
- FORMAT
- TITLE
- NAMED
- DEFAULT NULL
- COMPRESS USING and DECOMPRESS USING

For details on these data type attributes, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Data Definition Language Statements

The DATASET data type uses the SQL data definition language (DDL) statement, SET SESSION DOT NOTATION.

The DATASET data type enhanced the following statements:

- CREATE TABLE
- ALTER TABLE
- CREATE/REPLACE function
- CREATE storage_format SCHEMA
- SHOW storage_format SCHEMA
- DROP storage_format SCHEMA
- HELP storage_format SCHEMA
- CREATE INDEX
- COLLECT STATISTICS
- HELP, SHOW, and TYPE commands

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

Character Set Handling

The Avro specification requires that character strings be stored as UTF-8 encoded character data. Although the database does not support UTF-8 on the server side, it stores Avro character strings as such. When exporting a JSON-encoded Avro value, the Teradata UNICODE character set is used.

CSV values are stored in the same character set that was specified for the DATASET value.

About the DATASET Type CreateDATASET Function

Create an instance of the DATASET type through a cast or by using the CreateDATASET function.

About DATASET Type Transform

The DATASET type transform groups are used to import and export DATASET data from a client system to the database, and from the database to a client system.

You can specify which predefined transform group is used by default for a certain data type. The DATASET transform groups are based on the DATASET type instance storage format.

STORAGE FORMAT Avro transforms must meet the following requirements:

- Transform to/from BLOB using TD_DATASET_AVRO_BLOB (default).

The BLOB is composed of the UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value.

- Transform to/from VARBYTE using TD_DATASET_AVRO_VARBYTE

The VARBYTE is composed of the UTF-8 encoded schema, null-terminated, followed by the binary-encoded Avro value.

STORAGE FORMAT CSV transforms must meet the following requirements:

- Transform to/from CLOB in the CSV value character set using TD_Dataset_CSV_CLOB.

The CLOB is composed of a CSV value. Specified optional schemas are not included.

- Transforms to/from VARCHAR in the character set of the CSV value using TD_Dataset_CSV_VARCHAR.

The VARCHAR is composed of a CSV value. Specified optional schemas are not included.

The required formats of these transforms are equivalent to those defined for the system-defined casts.

You can use the TRANSFORM option in the CREATE PROFILE/MODIFY PROFILE or CREATE USER/MODIFY USER statements to specify for a user the particular transform group that will be used for a given data type.

Use the SET TRANSFORM GROUP FOR TYPE statement to change the active transform group in the current session. You can use this statement multiple times for a data type to switch from one transform group to another within the session. If the logon user already has transform settings, the statement modifies the transform settings for the current session.

Note:

You cannot use CREATE TRANSFORM or REPLACE TRANSFORM to create new transforms for complex data types (CDTs). You can only create new transforms for structured and distinct user-defined types (UDTs).

Transform Group Macros

You can use the following macros to find the transform group for a UDT (or CDT), or the transform group settings for a user, profile, or current session.

Macro	Description
SYSUDTLIB.HelpCurrentUserTransforms	Lists the transform group settings of the current logon user.
SYSUDTLIB.HelpCurrentSessionTransforms	Lists the transform group settings of the current session.
SYSUDTLIB.HelpUserTransforms(<i>User</i>)	Lists the transform group settings for a specific user.
SYSUDTLIB.HelpCurrentUDTTransform(<i>UDT</i>)	Lists the transform group settings of the current session for the specified UDT.
SYSUDTLIB.HelpUDTTransform(<i>User</i> , <i>UDT</i>)	Lists the transform group for a UDT for a user.

Macro	Description
SYSUDTLIB.HelpProfileTransforms(<i>Profile</i>)	Lists the transform group settings for a specific profile.
SYSUDTLIB. HelpProfileTransform(<i>Profile</i> , <i>UDT</i>)	Lists the transform group for a UDT for a profile.

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

About DATASET Type Cast

The DATASET data type implements the following casts:

- VARBYTE/BLOB to DATASET STORAGE FORMAT AVRO
- BYTE/VARBYTE/BLOB from DATASET STORAGE FORMAT AVRO
- VARCHAR/CLOB to DATASET STORAGE FORMAT CSV
- VARCHAR/CLOB from DATASET STORAGE FORMAT CSV
- DATASET STORAGE FORMAT CSV CHARACTER SET LATIN to DATASET STORAGE FORMAT CSV CHARACTER SET UNICODE
- DATASET STORAGE FORMAT CSV CHARACTER SET UNICODE to DATASET STORAGE FORMAT CSV CHARACTER SET LATIN

No other casts are supported.

When casting TO a DATASET data type for the STORAGE FORMAT AVRO or STORAGE FORMAT CSV, data must conform to the default specification. If not, use the CreateDATASET function to create a DATASET data type instance with either storage format Avro or CSV.

When casting FROM a DATASET data type for the STORAGE FORMAT AVRO, if the data is being cast to BYTE/VARBYTE/BLOB, the resulting data contains the schema defined for the instance, encoded in UTF-8 and null-terminated, followed by the binary-encoded Avro value.

When casting TO a DATASET STORAGE FORMAT CSV data type, the CSV schema is not associated with the data. Therefore, no validation is done, and the cast may result in mismatched data and schema. If you want to validate the data, use [CreateDATASET](#).

For STORAGE FORMAT CSV, if data is cast to VARCHAR/CLOB, the resulting data contains a CSV value. If a schema is associated with the CSV value, it is not included.

About DATASET Type Ordering

Ordering describes how data is ordered. For example, the order could be *ABCD*, where A is less than D.

The DATASET data type may not be ordered, compared, or grouped. A portion of the DATASET instance may be selected out by using the provided system methods or functions, or dot notation, and can be used in a comparison operation.

About DATASET Type Usage

User-defined Functions

Use the CREATE/REPLACE FUNCTION statement to create a user-defined function (UDF) containing one or more parameters, or a DATASET return type of any of supported storage format. The parameters and/or return types are supported on scalar, aggregate, and table UDFs, and SQL UDFs. When the return type is specified as one of these types for SQL UDF, the RETURN expression may be an SQL statement evaluating to one type.

The DATASET type as a parameter to a UDF is supported for LANGUAGE C, CPP, and JAVA, but LANGUAGE R is not supported.

The following action occurs on the Data Dictionary CREATE FUNCTION statement for the DATASET type, and is in addition to dictionary updates that normally occur on a CREATE FUNCTION statement:

- The row inserted to DBC.TVFields to record metadata information about the DATASET field indicates it is a DATASET type. It shares some entries with the UDTs. The FieldType is 'DT,' and the Typeld corresponds to the static type ID assigned to the DATASET type.

Table Operators

The DATASET type is supported in C language and Java language user-defined table operators. The metadata is passed to the table operator contract function using an external type code such as DATASET_AVRO_DT or DATASET_CSV_DT. The type codes for dtype_en include the following. See sqltypes_td.h for the complete definition.

```
typedef enum dtype_en
{
    UNDEF_DT=0,
    CHAR_DT=1,
    VARCHAR_DT=2,
    BYTE_DT=3,
    VARBYTE_DT=4,
    GRAPHIC_DT=5,
    VARGRAPHIC_DT=6,
    BYTEINT_DT=7,
    SMALLINT_DT=8,
    INTEGER_DT=9,
    BIGINT_DT = 36,
    REAL_DT=10,
    DECIMAL1_DT=11, '
    DECIMAL2_DT=12,
    DECIMAL4_DT=13,
    DECIMAL8_DT=14,
    DECIMAL16_DT=37,
```



```

DATE_DT=15,
TIME_DT=16,
TIMESTAMP_DT=17,
INTERVAL_YEAR_DT=18,
INTERVAL_YTM_DT=19,
INTERVAL_MONTH_DT=20,
INTERVAL_DAY_DT=21,
INTERVAL_DTH_DT=22,
INTERVAL_DTM_DT=23,
INTERVAL_DTS_DT=24,
INTERVAL_HOUR_DT=25,
INTERVAL_HTM_DT=26,
INTERVAL HTS_DT=27,
INTERVAL_MINUTE_DT=28,
INTERVAL_MTS_DT=29,
INTERVAL_SECOND_DT=30,
TIME_WTZ_DT=31,
TIMESTAMP_WTZ_DT=32,
BLOB_REFERENCE_DT=33,
CLOB_REFERENCE_DT=34,
UDT_DT=35,
/* The 8 byte integer type (BIGINT_DT) and
 * the 16 byte decimal type (DECIMAL16_DT)
 * are located above and have the following
 * values:
 *
 * BIGINT_DT=36
 * DECIMAL16_DT=37
 */
NUMBER_DT=38,
PERIOD_DT=39,
JSON_DT=40,
DATASET_AVRO_DT=41,
ST_GEOMETRY_DT=42,
MBR_DT=43,
MBB_DT=44,
ARRAY_DT=45,
XML_DT = 46,
    DATASET_CSV_DT=47,
    FNC_DATATYPESETSIZE=48
} dtype_en;

```

For AVRO, the complex types map to the following base type, DATASET_AVRO_DT → BLOB_REFERENCE_DT. For CSV, the complex types map to the following base type, DATASET_CSV_DT → CLOB_REFERENCE_DT

When input data values are sent to a table operator, the data is transferred in the current default transform. Each possible transform type populates the UDT_BaseInfo_t.transform_info structure, as shown in the following table:

Transform Type	Datatype	Column	...	Size.length
TD_DATASET_AVRO_VARBYTE	VARBYTE_DT	<name of the column>		The data size
TD_DATASET_AVRO_BLOB	BLOB_REFERENCE_DT	<name of the column>		The data size

For CSV, the transform types populate the UDT_BaseInfo_t.transform_info structure, as shown in the following table:

Transform Type	Datatype	Column	...	Charset	...	Size.length
TD_CSV_CLOB	CLOB_REFERENCE_DT	<name of the column>				The data size

There is no "transforms off" functionality. The UDT_BaseInfo_t structure's udt_indicator member value identifies the DATASET storage formats:

10==DATASET STORAGE FORMAT AVRO

11==DATASET STORAGE FORMAT CSV

For CSV, the UDT_BaseInfo_t's charset field determines the character set as either LATIN or UNICODE.

External Stored Procedures

The CREATE/REPLACE PROCEDURE statement was extended to create an external stored procedure containing one or more parameters that are DATASET types of any of the supported storage formats. Use these types to define the IN, OUT, or INOUT parameters.

The DATASET type as an IN, OUT, or INOUT parameter to an external stored procedure is supported for LANGUAGE C, CPP, or JAVA. The LANGUAGE R option is not supported.

The following action occurs on the Data Dictionary on a CREATE/REPLACE PROCEDURE statement for the DATASET type. The change is in addition to dictionary updates that normally occur on a CREATE/REPLACE PROCEDURE statement.

- The row normally inserted to DBC.TVFields to record metadata information about the DATASET field was enhanced to indicate that it is a DATASET type. It shares some entries with the UDTs. The FieldType is 'DT', and the TypeId corresponds to the static type ID assigned to the DATASET type.

FNC Library Routines That Support the DATASET Type

When developing UDFs or external stored procedures defined with DATASET type parameters or return values, use the following DATASET type interface functions to access or set the values of the DATASET type parameters, or to get information about a DATASET type instance.

FNC Library Routine	Description
FNC_GetDatasetInfo	Identifies the maximum length, in-line length, schema length, raw data length, whether the schema and/or data is a LOB, and storage format of any DATASET data type instance so users can write a generic routine to handle cases.
FNC_GetDatasetInputLob	Reads DATASET data stored as a LOB using the existing LOB FNC routines. The CSV value is returned in a LOB, but does not include an optional schema.
FNC_GetDatasetResultLob	Writes DATASET data to a LOB associated with any DATASET instance. The CSV value is written to the result LOB, but does not include an optional schema.
FNC_GetDatasetSchema	Retrieves the schema for any DATASET data type instance, regardless of storage format. The schema is returned as encoded in UTF-8 or UTF-16, depending on what the user specifies.
FNC_GetDatasetSchemaLob	Reads the schema of a DATASET instance stored as a LOB using the existing LOB FNC routines. The schema is returned as encoded in UTF-8 or UTF-16, depending on what the user specifies.
FNC_GetInternalValue	Retrieves non-LOB data from a DATASET instance. The CSV value is returned, but does not include an optional schema.
FNC_SetDatasetLob	Passes a LOB_LOCATOR that references a UTF-8 encoded schema, null-terminated, followed by the binary-encoded value to a DATASET data type instance. The data must conform to the transform format of the storage format of the DATASET instance.
FNC_SetInternalValue	Sets non-LOB data of a DATASET instance. When it is known that the data for a particular instance is not stored as a LOB, this routine can set the data. The data must conform to the transform format of the storage format of the DATASET instance. The CSV value is passed to this function, but it does not include an optional schema.

Use the DATASET_HANDLE data type to pass a DATASET type instance as an argument to an external routine or UDF. Similarly, use DATASET_HANDLE to return a DATASET type result from an external routine. DATASET_HANDLE is defined in sqltypes_td.h as follows:

```
typedef int DATASET_HANDLE;
```

Some FNC calls require that you specify encoding for the schema text being handled. Because of this, the following enum and types are defined in sqltypes_td.h:

```
typedef enum dataset_schema_encoding_en {
    datasetSchemaUTF8 = 0,
    datasetSchemaUTF16 = 1
} dataset_schema_encoding_en;
typedef Byte dataset_schema_encoding_t;
```

The CSV enum field in `dataset_storage_et` is called `DATASET_CSV_EN=1`.

The DATASET data type may not be used as an attribute of a structured UDT or as the base type of a Teradata Distinct UDT or ARRAY type.

For more information about the DATASET type interface functions, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Restrictions for the DATASET Type

There are length restrictions for the DATASET type.

A storage format must be specified. If not, the following error is reported:

```
Failure 3706 Syntax error: STORAGE FORMAT must be specified for the DATASET type.
```

Storage Format Restrictions

A schema must be specified, depending on the storage format used. A schema may be specified at the following levels:

- Column level. Specify a schema at the column level if included in the CREATE or ALTER TABLE statement as an attribute of a DATASET data type column AND the schema was previously registered using the CREATE *storage_format* SCHEMA statement.
- Instance level. Specify a schema at the instance level if no schema has been specified at a higher level. It can be specified to apply to multiple instances of the data (for example, when batch loading similar data) using the CreateDATASET function.

By default, data is validated against the schema provided for a particular instance. This can result in reduced performance. To disable this validation, set the DBS Control field `DisableDATASETValidation` to TRUE.

The DATASET data type may not be used as an attribute of a structured UDT or as the base type of a Teradata Distinct UDT or ARRAY type.

The DATASET data type may not be ordered, compared or grouped, so no ordering routine is provided.

Use the DATASET data type like any other CDT data type, except that it cannot be relationally compared. Therefore, it cannot be used in an index definition and cannot be used in comparison expressions. However, a portion of the DATASET instance may be selected out using provided system methods or functions. This portion may be used in a comparison operation within a query if it can be cast from a VARCHAR string to a comparable data type, compared as a VARCHAR, or be representative of another predefined, comparable type. Because the type may not be compared, it is not used for comparison in a SET table.

Operations on the DATASET Data Type

Creating and Altering Tables to Store DATASET Data

The CREATE TABLE statement supports the column-level attributes of the DATASET data type.

You can define DATASET table columns with column-level schemas or instance-level schemas.

Column-level Schemas

Usually, all DATASET values within a table column adhere to the same schema. In these cases, it makes sense to associate a schema with the table column. You can specify the schema using the CREATE *storage_format* SCHEMA statement, and you can associate the DATASET column with this schema using the WITH SCHEMA clause in the CREATE or ALTER TABLE statement.

There are storage and performance advantages when you use a column-level schema:

- The schema is stored once within the data dictionary. It is not stored with each DATASET value in the row. This can result in significant storage space savings and some minor performance savings due to less I/O.
- In many cases, the schema needs to be parsed only once for the DATASET column during a query execution. This can result in significant performance gains since schema parsing can be costly.

The following example shows a DATASET column in the Avro storage format with a column-level schema:

```
CREATE AVRO SCHEMA avro_schema_1 as '{"type":"record",
  "name":"rec_0","fields":[
    {"name":"ProductID","type":"int"},
    {"name":"Price","type":"int"}
  ]}';
```

```
CREATE TABLE mytable(c1 INTEGER, c2 DATASET STORAGE FORMAT AVRO WITH
SCHEMA avro_schema_1);
```

The following example shows a DATASET column in the CSV storage format with a column-level schema:

```
CREATE CSV SCHEMA myCSVSchema AS
'{
  "field_delimiter" : "\t",
  "record_delimiter" : ";"
}';
```

```
CREATE TABLE myDatasetTable02
(
```

```
id INTEGER,
csvFile DATASET(100000) INLINE LENGTH 5000 STORAGE FORMAT CSV CHARACTER SET
LATIN WITH SCHEMA myCSVSchema
);
```

Instance-level Schemas

If you omit the WITH SCHEMA clause in the CREATE or ALTER TABLE statement, the DATASET value is defined with an instance-level schema. In this case, both the schema and data are stored within the database row. This allows a DATASET table column to contain data values with different schemas. This flexibility comes at the cost of increased storage usage and reduced performance due to schema parsing.

The following shows an instance-level schema for a DATASET value with the CSV storage format:

```
CreateDATASET
(
  '{"field_delimiter": "&", "record_delimiter": "#"}',
  'Item ID&Item Name&Item Color #55&bicycle&red#88&toy boat&pink#105&soap&#',
  CSV,
  UNICODE
)
```

Disabling DATASET Validation

When DATASET values are inserted into DATASET table columns, the schema and the data value are validated, which can result in reduced performance. Even if the DATASET column is defined with a column-level schema, the data must be validated against the schema during insertion. You can disable this validation by setting the DBS Control field DisableDATASETValidation to TRUE.

Example: Creating a Table with the DATASET Data Type

To create a table with a DATASET data type, based on Avro, with a schema specification, use the following statement:

```
CREATE TABLE myDatasetTable03(
  id INTEGER,
  avroFile DATASET STORAGE FORMAT Avro WITH SCHEMA chemDatasetSchema
);
```

To create a table based on the storage format CSV, use this example:

```
CREATE TABLE csv_table(
  id INTEGER,
  csv DATASET STORAGE FORMAT CSV CHARACTER SET LATIN);
```

Using Algorithmic Compression on DATASET Columns

The storage format specification defines a file format that transmits and stores the values along with a common schema. It also defines certain processes for operating on and compressing the storage format data.

You can perform algorithmic compression on DATASET data using the SNAPPY_COMPRESS and SNAPPY_DECOMPRESS routines.

Note:

You can only use SNAPPY_COMPRESS and SNAPPY_DECOMPRESS to compress DATASET data. If you use any other algorithmic compression routines, you will get an error.

For storage format AVRO:

```
CREATE TABLE avroCompressTable(
  id INTEGER,
  compressibleAvroFile DATASET STORAGE FORMAT AVRO
    COMPRESS USING SNAPPY_COMPRESS
    DECOMPRESS USING SNAPPY_DECOMPRESS);
```

For storage format CSV:

```
CREATE TABLE csvCompressTable(
  id INTEGER,
  compressibleCSVFile DATASET STORAGE FORMAT CSV
    COMPRESS USING SNAPPY_COMPRESS
    DECOMPRESS USING SNAPPY_DECOMPRESS);
```

For more information about algorithmic compression, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

Accessing DATASET Data Using Dot Notation

Note:

This is a brief overview of using dot notation with the DATASET data type. It discusses dot notation as it relates to DATASET. For more information about dot notation, see *Teradata JSON*.

Dot notation supports the following members of the JSONPath syntax:

- Recursive descent operator (..)
- Wildcards (*) - both in reference to named and indexed items
- Name/index lists ([a,b,c] or [0,3,5])

- Name/index slices ([c] or [5])

The items are used for both the JSON and DATASET data types. The following examples and rules use these new syntax pieces in the SELECT list and the WHERE clause. Note that not all portions of the JSONPath syntax are supported by the DATASET types, including JSONPath expressions and filters.

The return value of a dot notation expression on a DATASET data type is VARCHAR by default. If the referenced DATASET type is a column of a table with a schema defined at the column level, the expected data type is inferred from the schema and used as the expression return type, if possible. There are certain scenarios where it is not possible. For example, if a dot notation expression retrieves multiple children of a record, which have different data types:

```
SELECT column.record[childA, childB, childC];
```

The following tables are referenced in examples throughout the book for the AVRO storage format:

```
CREATE TABLE myAVROTable09(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);

avroFiles09.txt
avro09.data|1

avro09.data
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473223
A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F7264222C
226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D65223A224974656D5F494
4222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C22747970
65223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F72222C2274797065223
A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C2274797065223A2273
7472696E67227D2C7B226E616D65223A225175616E746974795F507572636861736564222C2274797
065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C2274797065223A2264
6F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C2274797065223A22646F7
5626C65227D5D7D7D5D7D006E0E62696379636C650672656408626F79730200000000000059400000
000000005940

.import vartext file avroFiles09.txt
USING (c1 BLOB as deferred by name, c2 INTEGER)
INSERT INTO myAVROTable09(:c2,:c1);
```

If using the CSV storage format, substitute CSV in the examples instead:

```
CREATE TABLE myCSVTable09(
    id INTEGER,
    csvFile DATASET STORAGE FORMAT CSV);
```



```

csvFiles09.txt
csv09.data|1

csv09.data
ItemNum,ItemName,Quantity,Price
10,Basketball,15,24.99
20,Shovel,9,7.99
30,Paint Brush,10,3.99

.import vartext file csvFiles09.txt
USING (c1 CLOB as deferred by name, c2 INTEGER)
INSERT INTO myCSVTable09(:c2,:c1);

```

Example: Using Dot Notation with the DATASET Data Type

Example: Using the Table Myavrotable09 (AVRO):

```

/*simple named references on all rows in each instance*/

SELECT avroFile.Sale."Item_ID"
FROM Myavrotable09;
> 55

/*recursive descent operator on all rows in each instance*/

SELECT avroFile.."Item_ID"
FROM Myavrotable09;
> 55

/*wildcard operator on all rows in each instance*/

SELECT avroFile.Sale.*
FROM Myavrotable09;

> [55,"bicycle","red","boys",1,100.0,100.0]

/*named index list on all rows in each instance*/

SELECT avroFile.Sale["Item_ID","Item_Name"]
FROM Myavrotable09;
> [55,"bicycle"]

```

Example: Using the Table csv_table (CSV):

In this example, insert the following CSV data into the myCSVTable09 table. The first line contains the header, columns and records delimited by the default character.

```
ItemNum,ItemName,Quantity,Price
```

```
100,Paint Brush,3,7.99
```

```
101,Roller,5,12.99
```

```
102,Bucket,7,15.99
```

The CSV data is an array of records.

```
/* The following SELECT statement does not use the recursive descent operator
or the array syntax. Therefore, nothing is selected and null is returned. */
SELECT csv.ItemName from csv_table;
```

Result:

```
> ?
```

```
/* The field names are case sensitive, so in this example, "price" does
not match up with the field name of "Price". Therefore, the following query
returns null.*/
SELECT csv[*].price from csv_table;
```

Result:

```
> ?
```

```
/* Select a field from every record using the recursive descent operator. */
SELECT csv..ItemName from csv_table;
```

Result:

```
> ["Paint Brush","Roller","Bucket"]
```

```
/* Select two fields from every record using the recursive descent operator. */
SELECT csv..[ItemID,ItemName] from csv_table;
```

Result:

```
> ["100","Paint Brush","101","Roller","102","Bucket"]
```

```
/* Same as the previous example, but select the fields in a different order. */
SELECT csv..[ItemName,ItemID] from csv_table;
```

Result:

```
> ["Paint Brush","100","Roller","101","Bucket","102"]
```

```
/* Select record number 2 (all fields). */
SELECT csv[2] from csv_table;
```

Result:

```
> ["102","Bucket","7","15.99"]
```

```
/* Select all records (all fields). */
SELECT csv[*] from csv_table;
```

Result:

```
> ["100","Paint
Brush","3","7.99","101","Roller","5","12.99","102","Bucket","7","15.99"]
```

```
/* Select a field from every record using the array syntax. */
SELECT csv[*].Price from csv_table;
```

Result:

```
> ["7.99","12.99","15.99"]
```

```
/* Select out a field from record number 1. */
SELECT csv[1].ItemName from csv_table;
```

Result:

```
> Roller
```

```
/* Select all fields from record number 1. */
SELECT csv[1].* from csv_table;
```

Result:

```
> ["101","Roller","5","12.99"]
```

```
/* Select record 0 and record 2 (all fields) using the index list operator. */
SELECT csv[0,2] from csv_table;
```

Result:

```
> ["100","Paint Brush","3","7.99","102","Bucket","7","15.99"]
```

```
/* Select a field from record 0 and record 2 using the index list operator. */
SELECT csv[0,2].ItemName from csv_table;
```

Result:

```
> ["Paint Brush","Bucket"]
```

```
/* Select out record number 0 and 2 using the array slice operator with a step
value of 2 (so that record number 1 is skipped). */
SELECT csv[0:3:2] from csv_table;
```

Result:

```
> ["100","Paint Brush","3","7.99","102","Bucket","7","15.99"]
```

```
/* Select out a field from record number 0 and 2 using the array slice operator
with a step value of 2. */
SELECT csv[0:3:2].Price from csv_table;
```

Result:

```
> ["7.99","15.99"]
```

WHERE Clause Enhancements

With dot notation, users can compare search results for a JSON or DATASET type against a single operand. A list of results can be compared against a single operand using overloaded versions of the ANY/ALL/SOME clauses of the WHERE clause.

Example: Using the WHERE Clause

Example: Using the WHERE Clause (AVRO)

Using the table Myavrotable09:

```

/*determine if any sale included a bicycle*/

SELECT 'TRUE'
FROM Myavrotable09
WHERE 'bicycle' = ANY(avroFile.."Item_Name");
> TRUE

/*determine if any sale was for more than $50*/

SELECT 'TRUE'
FROM Myavrotable09
WHERE 50 < ANY(avroFile.."Total_Price");
> TRUE

/*determine if ALL sales were for more than $50*/

SELECT 'TRUE'
FROM Myavrotable09
WHERE 50 < ALL(avroFile.."Total_Price");
> TRUE

```

Example: Using the WHERE Clause (CSV)

Using the table MyCSVtable09:

```

/*determine if any sale included a basketball*/

SELECT 'TRUE'
FROM MyCSVtable09
WHERE 'Basketball' = ANY(csvFile.."ItemName");
> TRUE

/*determine if any sale was for more than $15*/

SELECT 'TRUE'
FROM MyCSVtable09
WHERE 15 < ANY(csvFile.."Total_Price");
> TRUE

/*determine if ALL sales were for more than $50*/

```

```
SELECT 'TRUE'
FROM MyCSVtable09
WHERE 50 < ALL(csvFile.."Total_Price");
*** No rows found
```

Modifying DATASET Columns

The INSERT-SELECT statement supports inserting a value to a DATASET data type column of any storage format. You can select and use a source table with a DATASET column whose data length is compatible with the DATASET data type column of the target table as a source value to the INSERT-SELECT operation. The source table must also have the same storage format as the target column.

If the source data is larger than the maximum possible length of the target DATASET data type column, an error occurs.

The data stored in the source table must contain a schema, so you do not need to specify a schema in the INSERT-SELECT statement. If you want the target table to store the data with an updated schema, the target column must have a schema specified. The schema specified in this location overwrites any schema specified for the source data, so be careful to ensure the new schema correctly describes the source data.

Examples

The examples use myAVROTable01 or MyCSVTable01, which were created and loaded as a source table.

For AVRO:

```
CREATE TABLE myAVROTable01(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
```

For CSV:

```
CREATE TABLE MyCSVTable01(
    id INTEGER,
    csvFile DATASET STORAGE FORMAT CSV CHARACTER SET LATIN);
```

Example: Successful INSERT-SELECT Statements

The following are examples of successful INSERT-SELECT statements.

For AVRO:

```
CREATE TABLE myAVROTable02(
    id INT,
```

```
        avroFile DATASET STORAGE FORMAT Avro);
INSERT INTO myAVROTable02 SELECT * FROM myAVROTable01;
```

For CSV:

```
CREATE TABLE MyCSVTable02(
    id INT,
    csvFile DATASET STORAGE FORMAT CSV CHARACTER SET LATIN);
INSERT INTO MyCSVTable02 SELECT * FROM MyCSVTable01;
```

Example: Unsuccessful INSERT-SELECT Statements

The following are examples of unsuccessful INSERT-SELECT statements, due to length incompatibilities.

For AVRO:

```
CREATE TABLE myAVROTable02(
    id INT,
    avroFile DATASET(100) STORAGE FORMAT Avro);
INSERT INTO myAVROTable02 SELECT * FROM myAVROTable01;
*** Failure 7548: The Avro value exceeds the maximum size of 100 specified for
this Dataset type
```

For CSV:

```
CREATE TABLE MyCSVTable02(
    id INT,
    csvFile DATASET(100) STORAGE FORMAT CSV);
INSERT INTO MyCSVTable02 SELECT * FROM myCSVTable01;
*** Failure 7548: The CSV value exceeds the maximum size of 100 specified for
this Dataset type
```

DATASET Methods, Functions, and Table Operators

You can perform the following common operations on the DATASET data type to access or manipulate DATASET data.

Methods

- AvroProject
- AvroProjectToJSON
- DATASET Constructor
- ExtractValue

- getRawData
- getRawDataLob
- getRawDataSize
- getSchema
- getSchemaSize
- Header
- numRecords
- toAvro
- toJSON
- Validate

Functions and Table Operators

- AVRO_CHECK
- AvroContainerSplit
- CreateDATASET
- CSVSplit
- CSV_TO_AVRO
- CSV_TO_JSON
- DATASET_KEYS
- DATASET_TABLE
- DataSize
- SchemaEqual
- SchemaMatch

DATASET Methods

AvroProject

AvroProject allows for reading specified portions of an Avro instance, and not the entire instance.

For example, take an Avro instance record with five fields: ID, First Name, Last Name, Phone Number, and Age.

To see just a few fields, such as First Name and Age, use AvroProject to project the source data into an Avro instance with a different schema composed of those fields.

AvroProject Syntax

```
DATASET_STORAGE_FORMAT_AVRO_expression.AvroProject ( schema_expression )
```

Syntax Elements

DATASET_STORAGE_FORMAT_AVRO_expression

Any expression that evaluates to a DATASET data type with STORAGE FORMAT AVRO.

schema_expression

Any expression that evaluates to a Teradata CHAR/VARCHAR/CLOB/JSON conforming to the Avro schema specifications.

Usage Notes

- The structure of the two schemas (one with all the fields, and one with just the First Name and Age fields) must be compatible. The SchemaMatch function compares CLOBs or JSON types representing an Avro schema for compatibility, and can confirm ahead of time whether the schemas are compatible.
- The resulting Avro instance is composed of the new schema with the fields (First Name and Age, in the example) and the projected binary-encoded Avro value. If the values for the Avro instance or the new schema are NULL, the result is NULL.
- If the new schema is invalid or incompatible with the original schema, an error occurs.

Example: Reading Specified Portions of an Avro Instance

```
CREATE TABLE avroTable(id INTEGER, avroCol DATASET STORAGE FORMAT AVRO);
/*insert some data composed of a record with five fields as mentioned above*/
/*
```

```

{"id":1,"First":"Leo","Last":"Tolstoy","Phone":"(800)-123-4657","Age":187}
*/
INSERT INTO avroTable(1,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A22466
9727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22747
97065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A22737
472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D0002064
C656F0E546F6C73746F791C28383030292D3132332D34363537F602'xb);
/*
{"id":2,"First":"Mark","Last":"Twain","Phone":"(800)-123-4657","Age":180}
*/
INSERT INTO avroTable(2,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A22466
9727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22747
97065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A22737
472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D0004084
D61726B0A547761696E1C28383030292D3132332D34363537E802'xb);
/*
{"id":3,"First":"William","Last":"Shakespeare","Phone":"(800)-123-4657","Age":45
1}
*/
INSERT INTO avroTable(3,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A22466
9727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22747
97065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A22737
472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D00060E5
7696C6C69616D165368616B657370656172651C28383030292D3132332D343635378607'xb);
/*
{"id":4,"First":"Charles","Last":"Dickens","Phone":"(800)-123-4657","Age":203}
*/
INSERT INTO avroTable(4,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A22466
9727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22747
97065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A22737
472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D00080E4
36861726C65730E4469636B656E731C28383030292D3132332D343635379603'xb);

/*now perform the projection*/
SELECT id,
avroCol.AvroProject('{"type":"record","name":"rec_0","fields":

```

```
[{"name":"First","type":"string"}, {"name":"Age","type":"int"}]}'
FROM avroTable ORDER BY id;
```

AvroProjectToJSON

AvroProjectToJSON is similar to the AvroProject method, but returns a JSON-encoded Avro value instead of an Avro instance as its result.

Result Type

NULL, if the values for the Avro instance or the new schema are NULL.

AvroProjectToJSON Syntax

```
DATASET_STORAGE_FORMAT_AVRO_expression.AvroProjectToJSON (
schema_expression )
```

Syntax Elements

DATASET_STORAGE_FORMAT_AVRO_expression

Any expression that evaluates to a DATASET data type with STORAGE FORMAT AVRO.

schema_expression

Any expression that evaluates to a Teradata CHAR/VARCHAR/CLOB/JSON conforming to the Avro schema specifications.

Usage Notes

The rules for AvroProject apply to AvroProjectToJSON, except that AvroProjectToJSON returns a JSON CHARACTER SET UNICODE instance.

Example: Returning a JSON-Encoded Avro Value Instead of an Avro Instance as its Result

```
SELECT id,
avroCol.AvroProjectToJSON('{"type":"record","name":"rec_0","fields":
[{"name":"First","type":"string"}, {"name":"Age","type":"int"}]}'
FROM avroTable ORDER BY id;
```

id	avroCol.AvroProjectToJSON(...)
----	--------------------------------

1	<code>{"First": "Leo", "Age": 187}</code>
2	<code>{"First": "Mark", "Age": 180}</code>
3	<code>{"First": "William", "Age": 451}</code>
4	<code>{"First": "Charles", "Age": 203}</code>

DATASET Constructor

The constructor method creates an instance of the DATASET data type in either the Avro or CSV storage format. Additionally, you can specify the character set for the CSV storage format.

DATASET Constructor Syntax

```
new DATASET (
  schema_expression,
  DATASET_expression,
  { AVRO | CSV [, { LATIN | UNICODE } ] }
)
```

Syntax Elements

schema_expression

Any expression that evaluates to a Teradata VARCHAR or VARBYTE conforming to the Avro or CSV schema specifications.

DATASET_expression

The CSV or Avro data value. If the constructor is for a CSV value, the value is a VARCHAR or CLOB in either the LATIN or UNICODE character set. If the constructor is for an Avro value, this value is VARBYTE or BLOB.

AVRO

CSV

The storage format of the new DATASET instance, either CSV or Avro.

LATIN UNICODE

The character set of the returned CSV DATASET value, either LATIN or UNICODE. You can specify the character set only for the CSV storage format. If this parameter is omitted for a CSV value, the return CSV value is created in the current server character set.

Examples

Example: Creating a CSV DATASET Value

The character set defaults to the current server character set.

```
SELECT new DATASET('{ "field_delimiter" : ",", "record_delimiter" : ";" }',
    'Id,Item,Qnty,Price;103,brush,33,0.99;102,paint,2,0.69',
    CSV);

> Id,Item,Qnty,Price;103,brush,33,0.99;102,paint,2,0.69
```

Example: Creating a CSV DATASET Value with a UNICODE Character Set

```
SELECT new DATASET('{ "field_delimiter" : ",", "record_delimiter" : ";" }',
    'Id,Item,Qnty,Price;103,brush,33,0.99;102,paint,2,0.69',
    CSV, Unicode);

> Id,Item,Qnty,Price;103,brush,33,0.99;102,paint,2,0.69
```

Creating an Avro DATASET Value

In this example, create an Avro DATASET value, and then select it out in the JSON format by using the `toJson()` method for readability.

```
SELECT new DATASET(
    '{ "type": "record",
      "name": "rec_0",
      "fields": [
        { "name": "ID", "type": "int" },
        { "name": "First", "type": "string" },
        { "name": "Middle", "type": "string" },
        { "name": "Last", "type": "string" } ] }',
    '081246726564657269636B064A6F6E1057696C6C69616D73'XB,
    Avro).toJson();

> { "ID": 4, "First": "Frederick", "Middle": "Jon", "Last": "Williams" }
```

ExtractValue

You cannot run all queries on a DATASET instance using dot notation. Some types include queries containing a dot notation expression too large to express in Teradata SQL, or queries using non-permitted characters or words in the syntax.

Instead, use ExtractValue for these queries.

ExtractValue Syntax

```
DATASET_expression.ExtractValue ('$dot_notation')
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

dot_notation

All dot notation elements are supported, including the following new elements:

- Recursive descent operator (..)
- Wildcards (*) - both in reference to named and indexed items
- Name/index lists ([a,b,c] or [0,3,5])
- Name/index slices ([c] or [5])

Usage Notes

You can use the ExtractValue method to evaluate any dot notation expression on any DATASET instance of any storage format. Using this method is recommended only when the desired query may not be properly expressed in dot notation, and is provided as a backup for such queries.

Examples

Example: Returning Values for a Query Not Properly Expressed in Dot Notation (AVRO)

Populate the myAVROTable06 table:

```
avro08b.data
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F726422
2C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D65223A224974656D5F
4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C2274
```

```

797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F7222C22747970
65223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C227479706522
3A22737472696E67227D2C7B226E616D65223A225175616E746974795F507572636861736564222C
2274797065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C2274797065
223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C227479706522
3A22646F75626C65227D5D7D7D5D7D006E0E62696379636C650672656408626F7973020000000000
0059400000000000005940
CREATE TABLE myAVROTable06(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
.import vartext file avro08b.data
USING (c1 VARBYTE(1000))
INSERT INTO myAVROTable06(1,:c1);

```

Then, return values by using the ExtractValue method. You also can use dot notation to express the return values.

```

SELECT avroFile.ExtractValue('$.Sale.Item_ID') FROM myAVROTable09;
> 55

SELECT avroFile.ExtractValue('$.Sale[Item_ID,Item_Name]') FROM
myAVROTable09;
> [55,"bicycle"]

```

Example: Returning Values for a Query Not Properly Expressed in Dot Notation (CSV)

Populate the myCSVTable06 table:

```

CREATE TABLE myCSVTable06(
    id INTEGER,
    csvFile DATASET STORAGE FORMAT CSV CHARACTER SET LATIN);
INSERT INTO myCSVTable06(
    0,
    new DATASET('{ "field_delimiter" : "&",
        "record_delimiter" : "#",
        "field_names" : ["Item_ID","Item_Name","Price"]}',
        '55&bicycle&89.99',
        CSV,
        LATIN));

```

Then, return values by using the ExtractValue method.

```
SELECT csvFile.ExtractValue('$[0].Item_ID') FROM myCSVTable06;
> 55

SELECT csvFile.ExtractValue('$..Item_ID,Item_Name') FROM myCSVTable06;
> "55","bicycle"
```

getRawData

getRawData retrieves the raw data of any DATASET data type instance, returning it as a non-LOB type. For the CSV storage format, the CSV file is returned, but not the schema.

Result Type

The data returned by a DATASET instance with STORAGE FORMAT AVRO or CSV conforms to the specification of the binary-encoded Avro or CSV values.

The return type is VARCHAR(64000) CHARACTER SET LATIN if the CSV value is in the LATIN character set, or VARCHAR(32000) CHARACTER SET UNICODE for UNICODE.

getRawData Syntax

```
DATASET_expression.getRawData ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Usage Notes

getRawData is only available to the DATASET data type. It provides the raw data of a DATASET data type instance, without including the schema.

Examples

Example: Retrieving the Raw Data of a DATASET Data Type Instance with the Avro Storage Format

```
SELECT cast(avroFile.getRawData() as varbyte(5000))
FROM myAVROTable06;
> 6E0E62696379636C650672656408626F797302
00000000000059400000000000005940
```


Example: Retrieving the Raw Data of a DATASET Data Type Instance with the CSV Storage Format

```
SELECT csvFile.getRawData() FROM myCSVTable06;
> 55&bicycle&89.99
```

getRawDataLob

getRawDataLob retrieves the raw data of any DATASET data type instance and returns it as a LOB type.

Result Type

The data returned by a DATASET instance with STORAGE FORMAT conforms to the specification of the binary-encoded Avro value, or to the CSV value.

The return type is CLOB CHARACTER SET LATIN if the CSV value is in the LATIN character set, or CLOB CHARACTER SET UNICODE for UNICODE.

getRawDataLob Syntax

```
DATASET_expression.getRawDataLob ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Usage Notes

The getRawDataLob is made available only to the DATASET data type. It provides the raw data of a DATASET data type instance, without including the schema.

Examples

Example: Retrieving the Raw Data of a DATASET Data Type Instance (AVRO)

```
getRawDataLob
SELECT avroFile.getRawDataLob() FROM myAVROTable06;
> 6E0E62696379636C650672656408626F797302
   00000000000059400000000000005940
```

Example: Retrieving the Raw Data of a DATASET Data Type Instance (CSV)

```
SELECT csvFile.getRawDataLob() FROM myCSVTable06;
> 55&bicycle&89.99
```

getRawDataSize

getRawDataSize retrieves the raw data size for a DATASET data type instance.

Result Type

The raw data provided without its schema for any DATASET data type instance, of any storage format. The result is in terms of bytes, not characters.

getRawDataSize Syntax

```
DATASET_expression.getRawDataSize ()
```

Syntax Elements***DATASET_expression***

Any expression that evaluates to a DATASET data type.

Examples

Example: Retrieving the Raw Data Size for a DATASET Data Type (AVRO)

```
SELECT avroFile.getRawDataSize() FROM myAVROTable06;
> 35
```

Example: Retrieving the Raw Data Size for a DATASET Data Type (CSV)

```
SELECT csvFile.getRawDataSize() FROM myCSVTable06;
> 16
```

getSchema

getSchema retrieves the schema of any DATASET data type instance.

Result Type

Provides a JSON text representation (max length, character set UNICODE) of the schema for a DATASET data type using any storage format.

getSchema Syntax

```
DATASET_expression.getSchema ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Examples

Example: Retrieving the Schema of an Avro Instance of the DATASET Data Type

```
SELECT avroFile.getSchema() FROM myAVROTable06;
> {
    "type": "record",
    "name": "rec_0",
    "fields": [
        {
            "name": "Sale",
            "type": {
                "type": "record",
                "name": "rec_1",
                "fields": [
                    {"name": "Item_ID", "type": "int"},
                    {"name": "Item_Name", "type": "string"},
                    {"name": "Item_Color", "type": "string"},
                    {"name": "Item_Style", "type": "string"},
                    {"name": "Quantity_Purchased", "type": "int"},
                    {"name": "Item_Price", "type": "double"},
                    {"name": "Total_Price", "type": "double"}
                ]
            }
        }
    ]
}
```

Example: Retrieving the Schema of a CSV Instance of the DATASET Data Type

```
SELECT csvFile.getSchema() FROM myCSVTable06;
```

```
> {
  "field_delimiter" : "&",
  "record_delimiter" : "#",
  "field_names" :
  [
    "Item_ID",
    "Item_Name",
    "Price"
  ]
}
```

getSchemaSize

getSchemaSize provides the size of the schema for any DATASET data type instance, of any storage format.

Result Type

The result is in terms of bytes, not characters. Because the schemas are encoded as UTF-8 when stored, this number represents the number of bytes in the UTF-8 encoding of the schema.

getSchemaSize Syntax

```
DATASET_expression.getSchemaSize ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Examples

Example: Retrieving the Schema Size for a DATASET Data Type Instance (AVRO)

```
SELECT avroFile.getSchemaSize() FROM myAVROTable06;
> 375
```

Example: Retrieving the Schema Size for a DATASET Data Type Instance (CSV)

```
SELECT csvFile.getSchemaSize() FROM myCSVTable06;
> 100
```

Header

Retrieves the header row of a DATASET data type with CSV storage format. Header only supports the CSV storage format.

Result Type

Available for the DATASET data type when the storage format employed is CSV, since Avro has no header. No header provided for the CSV file returns a null.

The return type is CLOB, in the character set of the CSV value.

Header Syntax

```
DATASET_STORAGE_FORMAT_CSV_expression.Header ()
```

Syntax Elements

DATASET_STORAGE_FORMAT_CSV_expression

Any expression that evaluates to a DATASET data type with the STORAGE FORMAT CSV.

Example: Retrieving the Header Row

```
SELECT csvFile.header() FROM myCSVTable09;
> ItemNum,ItemName,Quantity,Price
```

numRecords

Returns the number of records in the DATASET STORAGE FORMAT CSV value. If the value has a header, it is included in the record count. numRecords only supports the CSV storage format.

Result Type

numRecords is only available to the DATASET data type when the storage format is CSV.

The return value is INTEGER.

numRecords Syntax

```
DATASET_STORAGE_FORMAT_CSV_expression.numRecords ()
```

Syntax Elements

DATASET_STORAGE_FORMAT_CSV_expression

Any expression that evaluates to a DATASET data type with the STORAGE FORMAT CSV.

Example

```
SELECT csvFile.numRecords() FROM myCSVTable06;
> 1
```

toAvro

Converts any DATASET data type instance with the CSV storage format into an Avro instance.

Result Type

This method returns an Avro value. Similar to toJSON(), if a field is empty, the value is equal to the Avro null value. The toAvro return value is structured as an array of records.

toAvro Syntax

```
DATASET_expression.toAvro ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Example: Converting a DATASET Data Type Instance into an Avro Instance

```
SELECT csvFile.toAvro().toJSON() FROM myCSVTable06;
> [{"Item_ID":"55","Item_Name":"bicycle","Price":"89.99"}]
```

toJSON

toJSON converts any DATASET data type instance into a JSON text instance.

Result Type

toJSON returns a JSON-encoded Avro or CSV value. The character set of the resulting JSON object is always UNICODE.

CSV values can contain multiple records, so the JSON value is an array of objects where each object contains the fields in a CSV record. For empty fields, the value equals the JSON null value. The resulting JSON object is in the same character set as the CSV value.

If there is no header record or schema with field names, default CSV field names generate. For defined field names with a record containing more than fields, default field names generate for the extra fields in the record. For example, if three field names are defined and a record contains four fields, the fourth field is named csv_fld4.

toJSON Syntax

```
DATASET_expression.toJSON ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Examples

Example: Converting a DATASET Data Type Instance into a JSON Text Instance (AVRO)

```
SELECT avroFile.toJSON() FROM myAVROTable06;
>{
    "Sale" : {
        "Item_ID" : 55,
        "Item_Name" : "bicycle",
        "Item_Color" : "red",
        "Item_Style" : "boys",
        "Quantity_Purchased" : 1,
        "Item_Price" : 100.00,
        "Total_Price" : 100.00
    }
}
```

Example: Converting a DATASET Data Type Instance into a JSON Text Instance (CSV)

```
SELECT csvFile.toJSON() FROM myCSVTable06;

> [{"Item_ID":"55","Item_Name":"bicycle","Price":"89.99"}]
```

Validate

Invoke Validate on a DATASET type instance to report whether data is valid.

Result Type

The validation method returns an integer representing whether the DATASET type instance is valid or not. A 0 signifies an invalid instance; a 1 signifies a valid instance.

Parameter(s)	Return Type
None	INTEGER

Validate Syntax

```
DATASET_expression.Validate ()
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

Examples

Example: Validating a DATASET Type Instance (AVRO)

To create a table with a DATASET column and insert data with validation disabled, use the table 'myAVROTable06' and add one invalid row (the last 4 bytes of the Avro binary encoded value are missing):

```
avro08b.data
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647322
3A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F726422
2C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D65223A224974656D5F
4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C2274
797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F72222C22747970
65223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C227479706522
3A22737472696E67227D2C7B226E616D65223A225175616E746974795F507572636861736564222C
```



```
2274797065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C2274797065
223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C227479706522
3A22646F75626C65227D5D7D7D5D7D006E0E62696379636C650672656408626F7973020000000000
005940000000000000
/*DDL included as a reference*/
/*CREATE TABLE myAVROTable06(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);*/
.import vartext file avro08b.data
USING (avroData VARCHAR(10000), encoding VARCHAR(20))
INSERT INTO myAVROTable06(2, cast(TO_BYTES(:avroData,:encoding) AS DATASET
STORAGE FORMAT AVRO));

SELECT id, avroFile.validate() FROM myAVROTable06 ORDER BY 1;
```

id	avroFile.validate()
1	1
2	0

Example: Validating a DATASET Type Instance (CSV)

```
SELECT id, csvFile.validate() FROM myCSVTable06 ORDER BY 1;
```

id	csvFile.validate()
0	1

DATASET Functions and Operators

AVRO_CHECK

The AVRO_CHECK function allows you to validate BYTE/VARBYTE/BLOB/DATASET STORAGE FORMAT AVRO data using the Avro specification, and to check the data validity.

AVRO_CHECK Syntax

```
AVRO_CHECK ( Avro_transform_format_data )
```

Syntax Elements

Avro_transform_format_data

Any expression that evaluates to Teradata BYTE/VARBYTE/BLOB/DATASET STORAGE FORMAT AVRO data conforming to the Avro schema specifications.

Usage Notes

- Only BYTE/VARBYTE/BLOB/DATASET STORAGE FORMAT AVRO data can be validated with the AVRO_CHECK function; other data types cannot be validated.
- The schema is validated first, then the data is validated against the schema. Any errors encountered are reported back using the return value.

Examples

Example: Storing Data In a Table

```
SELECT AVRO_CHECK(avroFile) FROM myAVROTable09;  
> OK
```

Example: Passing Valid Data

In this example, valid data is passed in as a constant VARBYTE.

```
SELECT AVRO_CHECK  
( '002274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
```

```
223A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002'xb);
> Invalid Avro Schema: Unexpected end-of-input
```

Example: Passing Invalid Data

The following example shows invalid data passed in as a constant VARBYTE.

```
SELECT
AVRO_CHECK('002274797065223A227265636F7264222C226E616D65223A227265635F30222C2266
69656C6473223A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002'xb);
> Invalid Avro Schema: Unexpected end-of-input
```

AvroContainerSplit

The AvroContainerSplit operator splits a BLOB in the Avro Object Container File format into a table of individual Avro DATASET objects.

AvroContainerSplit Syntax

```
AvroContainerSplit ( ON
  ( SELECT ( container_id, container )
    ( select_stmt_options )
  )
)
```

Syntax Elements

container_id

Uniquely identifies the Avro Object Container File, useful when processing multiple container files.

container_id includes the following types:

- BYTEINT
- SMALLINT
- INTEGER
- BIGINT
- DOUBLE PRECISION
- DECIMAL
- NUMBER
- CHAR

- VARCHAR

container

A BLOB in the Avro Object Container File format.

An error is returned if the input data is not one of these data types, or if the input rows contain more or less than two columns. If the container value is a null value, a row with an *avro_object_id* and *avro_value* set to null is returned. The result of `AvroContainerSplit` is one or more output rows, where each row represents an individual Avro value from the Object Container File. Three output columns are returned as follows:

- *out_container_id*: The same type and value as the *container_id* input column. *out_container_id* identifies the Object Container File that the Avro value came from.
- *avro_object_id*: Each Avro value from a container is numbered from 0 to $n-1$, where n is the number of Avro values within the Object Container File. The output column is type INTEGER.
- *avro_val*: The Avro Dataset value.

The Object Container File can be uncompressed (null codec) or compressed (deflate codec). The schema within the Object Container File is limited to 16 MB, and the uncompressed size of any file data block within the Object Container file is also limited to 16 MB.

select_stmt_options

The allowable or necessary options in Vantage for a SQL SELECT statement.

Example: Using AvroContainerSplit

The following example uses `AvroContainerSplit` on a table of BLOBs representing files in the Avro Object Container File format.

```
/* Create a table that contains the container objects */
CREATE TABLE containers(id INTEGER, container BLOB);

/* Load one or more containers into the containers table via
   the utility of your choice. */

/* Create a table that will receive the individual Avro objects
   from the AvroContainerSplit table operator. */
CREATE TABLE avro_table(
  container_id INTEGER,
  avro_id      INTEGER,
  avro_val     DATASET(8000) STORAGE FORMAT AVRO);

/* Invoke the table operator to split up the container. */
```

```

INSERT INTO avro_table
SELECT T.out_container_id, T.avro_object_id, T.avro_value
FROM AvroContainerSplit (ON (SELECT id, container FROM containers)) T;

/* Select out the individual Avro values in the JSON format. */
SELECT container_id, avro_id, avro_val.ToJson()
FROM avro_table
ORDER BY container_id, avro_id;

```

CreateDATASET

Use CreateDATASET to create DATASET data type instances composed of self-describing data when the schema and data are not included in the data payload. If the data is already with its schema, you do not need CreateDATASET to create an instance from the data.

CreateDATASET Syntax

```

CreateDATASET (
  schema_expression,
  data_expression,
  { AVRO | CSV [ , { LATIN | UNICODE } ] }
)

```

Syntax Elements

schema_expression

Any expression that evaluates to the following Teradata data types conforming to the schema specifications:

- CHAR
- VARCHAR
- CLOB
- JSON
- BYTE
- VARBYTE
- BLOB

If the Schema parameter is null, it is assumed the instance is being loaded into a column with a column-based schema. If not, and you use the instance without providing a schema (for example, writing the instance out to a table, or searching using dot notation), an error occurs.

The schema data is provided as a CHAR/VARCHAR/CLOB in either the LATIN or UNICODE character set, or as a BYTE/VARBYTE/BLOB representing the UTF8 encoding of the schema. The schemas provided as BYTE/VARBYTE/BLOB data are not null-terminated.

data_expression

Any expression that evaluates to Teradata BYTE/VARBYTE/BLOB/CLOB data.

A BYTE/VARBYTE/BLOB containing the binary-encoded self-describing data value. Data can be specified in any way Vantage currently allows for function parameters. If validation is enabled, and a null value does not complete the specified schema, an error occurs. Otherwise, the data is accepted.

AVRO CSV

Specify the name of the storage format. Any other value, including a null or omitted value, are invalid.

LATIN UNICODE

The character set of the created CSV DATASET value, either LATIN or UNICODE. You can specify the character set only for the CSV storage format. If this parameter is omitted for a CSV value, the CSV value is created in the default server character set.

Examples

Example: Creating DATASET Data Type Instances

In this AVRO example, a DATASET data type instance is created where the schema and data are not included in the data payload.

```
CREATE TABLE nonStandardAVRO(id INTEGER, avroFile DATASET STORAGE FORMAT Avro);

/*Load Avro schema and data separately*/
avroSchemaAndData.txt
7B2274797065223A226172726179222C226974656D73223A5B7B2274797065223A227265636F7264
222C226E616D65223A227265635F30222C226669656C6473223A5B7B226E616D65223A2253616C65
222C2274797065223A7B2274797065223A227265636F7264222C226E616D65223A227265635F3122
2C226669656C6473223A5B7B226E616D65223A224974656D5F4944222C2274797065223A22696E74
227D2C7B226E616D65223A224974656D5F4E616D65222C2274797065223A22737472696E67227D2C
7B226E616D65223A224974656D5F436F6C6F72222C2274797065223A22737472696E67227D2C7B22
6E616D65223A224974656D5F5374796C65222C2274797065223A22737472696E67227D2C7B226E61
```

```

6D65223A225175616E746974795F507572636861736564222C2274797065223A22696E74227D2C7B
226E616D65223A224974656D5F5072696365222C2274797065223A22646F75626C65227D2C7B226E
616D65223A22546F74616C5F5072696365222C2274797065223A22646F75626C65227D5D7D7D5D7D
2C7B2274797065223A227265636F7264222C226E616D65223A227265635F32222C226669656C6473
223A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F7264
222C226E616D65223A227265635F33222C226669656C6473223A5B7B226E616D65223A224974656D
5F4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C22
74797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F72222C227479
7065223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C2274797065
223A226E756C6C227D2C7B226E616D65223A225175616E746974795F507572636861736564222C22
74797065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C227479706522
3A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C2274797065223A
22646F75626C65227D5D7D7D5D7D2C7B2274797065223A227265636F7264222C226E616D65223A22
7265635F34222C226669656C6473223A5B7B226E616D65223A2253616C65222C2274797065223A7B
2274797065223A227265636F7264222C226E616D65223A227265635F35222C226669656C6473223A
5B7B226E616D65223A224974656D5F4944222C2274797065223A22696E74227D2C7B226E616D6522
3A224974656D5F4E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2249
74656D5F436F6C6F72222C2274797065223A226E756C6C227D2C7B226E616D65223A224974656D5F
5374796C65222C2274797065223A226E756C6C227D2C7B226E616D65223A225175616E746974795F
507572636861736564222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F50
72696365222C2274797065223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072
696365222C2274797065223A22646F75626C65227D5D7D7D5D7D|
0600E0E62696379636C650672656408626F79730200000000000059400000000000005940026E10
746F7920626F61740870696E6B02333333333332E403333333333332E4004D20108736F617002AE
47E17A14AEEF3FAE47E17A14AEEF3F00|1

```

```

.import vartext file avroSchemaAndData.txt
USING (c1 VARBYTE(10000), c2 VARBYTE(1000), c3 VARCHAR(10))
INSERT INTO nonStandardAVRO(:c3,CreateDATASET(:c1, :c2, Avro));

```

```

/*Retrieve the loaded data using the toJSON method*/
SELECT avroFile.toJSON() FROM nonStandardAVRO;

```

```

avroFile
-----
> [
    {"rec_0" : {"Sale" : {
        "Item_ID" : 55,
        "Item_Name" : "bicycle",
        "Item_Color" : "red",
        "Item_Style" : "boys",
        "Quantity_Purchased" : 1,
        "Item_Price" : 100.00,
        "Total_Price" : 100.00
    }
    }
}

```

```

    }},
    {"rec_2" : {"Sale" : {
        "Item_ID" : 55,
        "Item_Name" : "toy boat",
        "Item_Color" : "pink",
        "Item_Style" : null,
        "Quantity_Purchased" : 1,
        "Item_Price" : 15.10,
        "Total_Price" : 15.10
    }},
    {"rec_4" : {"Sale" : {
        "Item_ID" : 105,
        "Item_Name" : "soap",
        "Item_Color" : null,
        "Item_Style" : null,
        "Quantity_Purchased" : 1,
        "Item_Price" : 0.99,
        "Total_Price" : 0.99
    }},
    }},
    ]

```

The following example is for CSV:

```

CREATE TABLE nonStandardCSV(id INTEGER, csvFile DATASET STORAGE FORMAT CSV);

/*Create a DATASET data type instance of CSV data with non-standard delimiters
in the LATIN character set. */

INSERT INTO nonStandardCSV (1, CreateDATASET('{"field_delimiter": "&",
"record_delimiter": "#"}', 'Item ID&Item Name&Item Color&Item Style&Quantity
Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99', CSV, LATIN));

/*Create a DATASET data type instance of CSV data with no header line in the
LATIN character set. */

INSERT INTO nonStandardCSV (2, CreateDATASET(
'{"field_delimiter": "&", "record_delimiter": "#", "field_names": ["Item ID",
"Item Name", "Item Color", "Item Style", "Quantity Purchased",
"Item Price", "Total Price"]}', '55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99', CSV, LATIN ));

/*Create a DATASET data type instance of CSV data with character set
UNICODE. */

```



```
INSERT INTO nonStandardCSV (3, CreateDATASET('{ "field_delimiter": "&",
"record_delimiter": "#" }', 'Item ID&Item Name&Item Color&Item Style&Quantity
Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99', CSV, UNICODE));
```

```
/*Create a DATASET data type instance of CSV data where some fields are
enclosed in double quotes in the UNICODE character set*/
```

```
INSERT INTO nonStandardCSV (4, CreateDATASET('{ "field_delimiter": "&",
"record_delimiter": "#"}', 'Item ID&Item Name&Item Color&Item Style&Quantity
Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&"soap&shampoo combo"&&&1&0.99&0.99',
CSV, UNICODE));
```

```
/*Retrieve these rows of data that were inserted*/
```

```
SELECT * FROM nonStandardCSV ORDER BY id;
```

ID	csvFile
1	Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
2	Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
3	Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
4	Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total Price#55&bicycle&red&boys&1&100.00&100.00#88&toy boat&pink&&1&15.10&15.10#105&"soap&shampoo combo"&&&1&0.99&0.99

Example: Using Column Based Schema that Loads Several Rows of Data

```
CREATE AVRO SCHEMA avroSaleSchema AS
'{
  "type": "record",
  "name": "rec_0",
  "fields": [
```

```

    {
        "name": "Sale",
        "type": {
            "type": "record",
            "name": "rec_1",
            "fields": [
                {"name": "Item_ID", "type": "int"},
                {"name": "Item_Name", "type": "string"},
                {"name": "Item_Color", "type": "string"},
                {"name": "Item_Style", "type": "string"},
                {"name": "Quantity_Purchased", "type": "int"},
                {"name": "Item_Price", "type": "double"},
                {"name": "Total_Price", "type": "double"}
            ]
        }
    }
}];

```

```

CREATE TABLE avroSaleTable (
    id INTEGER,
    saleInfo DATASET STORAGE FORMAT AVRO WITH SCHEMA avroSaleSchema);

```

avrosaledata.data

```

6E0E62696379636C650672656408626F797302000000000000594000000000005940|1
6E0E62696379636C650672656408626F797302000000000000594000000000005940|2
6E0E62696379636C650672656408626F797302000000000000594000000000005940|3
6E0E62696379636C650672656408626F797302000000000000594000000000005940|4
6E0E62696379636C650672656408626F797302000000000000594000000000005940|5
6E0E62696379636C650672656408626F797302000000000000594000000000005940|6
6E0E62696379636C650672656408626F797302000000000000594000000000005940|7
6E0E62696379636C650672656408626F797302000000000000594000000000005940|8
6E0E62696379636C650672656408626F797302000000000000594000000000005940|9
6E0E62696379636C650672656408626F797302000000000000594000000000005940|10

```

```

.import vartext file avrosaledata.txt
USING (c1 VARBYTE(1000), c2 INTEGER)
INSERT INTO avroSaleTable(cast:id AS INTEGER),CreateDATASET(null,
TO_BYTES(:AvroData, :encoding), Avro));

```

```

/*retrieve the loaded data using the toJSON method*/
SELECT id, saleInfo.toJSON() FROM avroSaleTable WHERE id = 1;

```

id	avroCol.AvroProject(...)
1	<pre>{ "Sale" : { "Item_ID" : 55, "Item_Name" : "bicycle", "Item_Color" : "red", "Item_Style" : "boys", "Quantity_Purchased" : 1, "Item_Price" : 100.00, "Total_Price" : 100.00 } }</pre>

CSVSplit

Splits VARCHAR or CLOB with a CSV format or DATASET STORAGE FORMAT CSV value into a table of DATASET objects. The objects are in CSV storage format with one record per object.

By specifying a DATASET STORAGE FORMAT CSV value as the input parameter, you can convert CSV values containing multiple records into CSV values containing a single record.

CSVSplit Syntax

```
CSVSplit ( ON
  ( SELECT csv_text select_stmt_options )
  [ RETURNS (
    'data' DATASET STORAGE FORMAT CSV
    [ CHARACTER SET [ UNICODE | LATIN ] ]
  )
]
[ USING SCHEMA ( schema ) ]
)
```

Syntax Elements

csv_text

Any text that evaluates to character data in the CSV format.

select_stmt_options

The allowable or necessary options in Vantage for a SQL SELECT statement.

RETURNS

Specifies the output data type or length. The default is to return a maximum length CSV, so omit RETURNS for this output.

DATASET STORAGE FORMAT CSV

Returns data that uses the CSV storage format. The default returns a CSV of maximum length, so the RETURNS clause may be omitted if this is the desired output.

CHARACTER SET

The character set specified by DATASET STORAGE FORMAT CSV. The options include UNICODE or LATIN.

SCHEMA

Specify SCHEMA to explicitly define the published data structure. Use the clause with a character string representing an ad-hoc schema specification; any other value results in an error. If an ad hoc schema is specified, the structure must conform to the CSV format rules. If the clause is not specified, the input CSV data is assumed to conform to the defaults.

Usage Notes

The output value is always a single record with no header in the CSV data. Each output DATASET value includes a schema if any of the following occurs:

- The VARCHAR/CLOB CSV file has a header record
- The input value is a DATASET value with a schema
- The USING SCHEMA clause specifies a schema

The output DATASET values do not include the header record from the input file, but the *field_names* key places the header names in the schema.

Use CSVSplit to populate a destination table with output DATASET values. Because the values adhere to the same schema, use a schema associated with the destination table column by WITH SCHEMA so the schema is not stored in every row. The schema included with the output DATASET value is discarded if a schema is already defined for the table column. The table column schema must match the schema in the output DATASET value.

CSVSplit uses the following clauses to compose DATASET data type instances:

- SCHEMA
- RETURNS

Examples

The following example uses CSVSplit on a table of CLOBs representing CSV text files.

Example: Creating a Table Containing CSV Files

```
CREATE TABLE CSVFiles(id INTEGER, csvFile CLOB);
```

Example: Inserting CSV Data into the Table

```
INSERT INTO CSVFiles VALUES(0, 'ItemID,ItemName,Quantity,Price;10021,Paint
Brush,10,10.99;10033,Paint,3,24.99');
```

Example: Creating a Table for Split Operation Results

```
CREATE MULTiset TABLE csv_table(
file_id INTEGER,
csv DATASET(8000) STORAGE FORMAT CSV CHARACTER SET LATIN);
```

Example: Invoking the Table Operator to Split up CSV Files

```
INSERT INTO csv_table
SELECT id, data FROM CSVSplit
(
ON (SELECT csvFile, id FROM CSVFiles)
RETURNS(data DATASET STORAGE FORMAT CSV CHARACTER SET LATIN)
USING SCHEMA('{ "record_delimiter": ";" }')
) AS T;
```

Example: Selecting out Individual CSV Values

```
SELECT file_id, csv
FROM csv_table
ORDER BY file_id;

> file_id csv
    0 10033,Paint,3,24.99
    0 10021,Paint Brush,10,10.99
```

DATASET_KEYS

The DATASET_KEYS table operator provides a list of all keys that can be queried in a DATASET data type instance, or a DATASET schema specified as CHAR/VARCHAR/CLOB/JSON.

DATASET_KEYS Syntax

```
DATASET_KEYS ( ON
  ( SELECT DATASET_expression select_stmt_options )
  [ USING QUOTES ( { 'Y' | 'N' } ) ]
)
```

Syntax Elements

DATASET_expression

Any expression that evaluates to a DATASET data type.

select_stmt_options

The *select_stmt_options* are the allowable or necessary options in Vantage for a SQL SELECT statement.

USING QUOTES

Variables used as input to the SQL statement specified by *select_stmt_options*. The USING clause options available for the DATASET type are Y (Yes) or N (No).

Usage Notes

The input data to the DATASET_KEYS table operator is determined by the result of the SELECT statement in the ON clause. The input data must be one or more rows of data with one column where the data type is either:

- DATASET
- CHAR/VARCHAR/CLOB/JSON/VARBYTE/BYTE/BLOB and the data represents a valid schema for the DATASET type storage format in use.

If the input data is not one of those data types, or the schema data passed in is invalid, an error occurs. If the data passed in consists of zero rows, a null value is returned.

The result of the table operator is one or more output rows, where each row represents a path that can be queried in either the DATASET instance or schema. You can specify an optional parameter, QUOTES, in the USING clause of the table operator. The results are wrapped in quotes.

Note:

DATASET_KEYS can only parse input data that is equal to or less than the maximum size defined for the return value.

For example, if the input data is a CSV value of length 32,001 but the return value size is defined as 32,000, then DATASET_KEYS cannot parse the input even though the actual value that DATASET_KEYS will return may be less than 32,000.

You can use the DATASETAttributeSize DBS Control field to change the default return value size.

Example: Using DATASET_KEYS

The following example uses DATASET_KEYS on Avro data already stored in a table.

```
/*simple key extraction*/
SELECT * FROM DATASET_KEYS
(
    ON (SELECT avroFile FROM myAVROTable09)
) AS avroKeys ORDER BY 1;
> Item_ID
  Item_Name
  Item_Color
  Item_Style
  Quantity_Purchased
  Item_Price
  Total_Price

/*display the keys wrapped in double quotes*/
SEL * FROM DATASET_KEYS
(
    ON (SELECT avroFile FROM myAVROTable09)
    USING QUOTES(Y)
) AS avroKeys ORDER BY 1;
> "Item_ID"
  "Item_Name"
  "Item_Color"
  "Item_Style"
  "Quantity_Purchased"
  "Item_Price"
  "Total_Price"
```

This example uses DATASET_KEYS on CSV data already stored in a table.

```

/*simple key extraction*/
SELECT * FROM DATASET_KEYS
(
  ON (SELECT csvFile FROM myCSVTable09)
) AS csvKeys ORDER BY 1;
> ItemName
  ItemNum
   Price
  Quantity

/*display the keys wrapped in double quotes*/

SELECT * FROM DATASET_KEYS
(
  ON (SELECT csvFile FROM myCSVTable09)
  USING QUOTES('Y')
) AS csvKeys ORDER BY 1;
> "ItemName"
  "ItemNum"
   "Price"
  "Quantity"

```

DATASET_TABLE

The DATASET_TABLE table operator takes a DATASET instance and creates a temporary table based on a subset (or all) of the data contained within.

DATASET_TABLE Syntax

```

DATASET_TABLE ( ON
  ( DATASET_instances_retrieving_expression )
  [ USING ( { row_expression_literal | column_expression_literal } ) ]
)

```

Note:

Only the AVRO storage format uses the *row_expression_literal* in the USING clause. CSV does not.

Syntax Elements

Dataset_instances_retrieving_expression

A query expression, table name, or view name that results in shredding DATASET instances.

There must be at least two result columns. One column identifies a problematic DATASET instance, if encountered. The other column is the DATASET instance itself. You can include other columns that follow the DATASET instance column.

For information on creating a table, see [Examples](#) in the "Modifying DATASET Columns" section.

A typical example:

```
SELECT id, DatasetCol.toJson() FROM my_table ORDER BY 1;
```

An example with extra columns:

```
SELECT id, orderDataset, orderDate, orderSite FROM orderDatasetTable;
```

Extra columns are output as extra columns returned by the table operator.

If the *Dataset_instances_retrieving_expression* parameter is null, the function results in a table with no rows.

USING

Input to the table operator as specified by *Dataset_instances_retrieving_expression*.

row_expression_literal

A LATIN or UNICODE string literal conforming to the dot notation syntax supported on the DATASET type. It must use the '\$' prefix at the beginning, which represents the root of the DATASET instance. This parameter defines the set of data that creates the output rows.

If this parameter is null, an error occurs.

This expression applies only to the AVRO storage format. The CSV format does not use *row_expression_literal*.

column_expression_literal

A LATIN or UNICODE string literal conforming to the DATASET type dot notation syntax. It must use the '\$' prefix at the beginning, representing the DATASET instance root, or the row expression result (depending on if the 'fromRoot' attribute is present). *column_expression_literal* defines output table columns, and which data populates the columns.

If this parameter is null, an error occurs.

Usage Notes

DATASET_TABLE resides in TD_SYSFNLIB. The query result ID column is a number or character type, excluding CLOB, and must be unique.

The result is subject to a maximum row size, and the query cannot exceed the maximum allowable query size.

column_expression_literal requires mapping the RowExpr columns to the function output table columns. The output table column data type is any non-LOB predefined Teradata type specified in the *column_expression_literal*. Each column in *column_expression_literal* is defined by a JSON object conforming to normal column or ordinal column.

Example of a normal column:

```
{ "dotnotation" : "<path to source data for output column>",
  "type" : "<data type of output column>",
  "fromRoot" : true (if search is to be done from the root, instead of
the row expression result) }
```

Example of an ordinal column:

```
{ "ordinal" : true }
```

For CSV files, default assignment for the source column to target table columns is based on position. The first CSV column is assigned to the first table column, the second CSV column is assigned to the second table column, and so on. The CSV data does not have to align with the target table being loaded. Since CSV data is shredded into a temporary table, use the optional method to INSERT or SELECT between tables with columns not aligning perfectly.

Examples

Example: Using the Built-In Cast from Varbyte to DATASET to Perform the Insert

The data in its text JSON format is shown further on, to aid in understanding the data.

The example uses the storage format AVRO.

```
CREATE TABLE my_table (id INTEGER, DatasetCol DATASET STORAGE FORMAT AVRO);
INSERT INTO my_table (1,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D6
5223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73222
C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B2274797065223A2
27265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D652
23A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797065222
C2274797065223A22737472696E67227D5D7D7D2C7B226E616D65223A226A6F62222C227479706
5223A22737472696E67227D5D7D00E43616D65726F6E3008084C616B6514656C656D656E7461727
90E4D616469736F6E0C6D6964646C650C52616E63686F0868696768065543490E636F6C6C6567650
```

```

01470726F6772616D6D6572'xb);
INSERT INTO my_table (2,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D6
5223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73222
C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B2274797065223A2
27265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D652
23A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797065222
C2274797065223A22737472696E67227D5D7D7D7D5D7D000E4D656C697373612E08084C616B65146
56C656D656E746172790E4D616469736F6E0C6D6964646C650C52616E63686F0868696768144D697
26120436F7374610E636F6C6C65676500'xb);
INSERT INTO my_table (3,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D6
5223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73222
C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B2274797065223A2
27265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D652
23A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797065222
C2274797065223A22737472696E67227D5D7D7D7D2C7B226E616D65223A226A6F62222C227479706
5223A22737472696E67227D5D7D0008416C65783208084C616B6514656C656D656E746172790E4D6
16469736F6E0C6D6964646C650C52616E63686F08686967680A43535534D0E636F6C6C656765000
6435041'xb);
INSERT INTO my_table (4,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D6
5223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73222
C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B2274797065223A2
27265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D652
23A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797065222
C2274797065223A22737472696E67227D5D7D7D7D2C7B226E616D65223A226A6F62222C227479706
5223A22737472696E67227D5D7D000A44617669643206084C616B6514656C656D656E746172790E4
D616469736F6E0C6D6964646C650C52616E63686F08686967680028736D616C6C20627573696E657
373206F776E6572'xb);

SELECT id, DatasetCol.toJson() FROM my_table ORDER BY 1;

*** Query completed. 4 rows found. 2 columns returned.
*** Total elapsed time was 1 second.

      id DatasetCol.TOJSON()
-----
      1 {"name":"Cameron","age":24,"schools":
[{"name":"Lake","type":"elementary"}, {"name":"Madison","type":"middle"},
{"name":"Rancho","type":"high"},

```

```
{ "name": "UCI", "type": "college" } ], "job": "programmer" }
  2 { "name": "Melissa", "age": 23, "schools":
[ { "name": "Lake", "type": "elementary" }, { "name": "Madison", "type": "middle" },
{ "name": "Rancho", "type": "high" }, { "name": "Mira Costa", "type": "college" } ] }
  3 { "name": "Alex", "age": 25, "schools":
[ { "name": "Lake", "type": "elementary" }, { "name": "Madison", "type": "middle" },
{ "name": "Rancho", "type": "high" }, { "name": "CSUSM", "type": "college" } ], "job": "CPA" }
  4 { "name": "David", "age": 25, "schools":
[ { "name": "Lake", "type": "elementary" }, { "name": "Madison", "type": "middle" },
{ "name": "Rancho", "type": "high" } ], "job": "small business owner" }
```

```
SELECT * FROM DATASET_Table (
  ON (SELECT id, DatasetCol FROM my_table WHERE id=1)
  USING rowexpr('$schools[*]')
  colexpr(
    '[ { "dotnotation" : "$.name",
      "type" : "CHAR(20)",
      { "dotnotation" : "$.type",
        "type" : "VARCHAR(20)",
        { "dotnotation" : "$.name",
          "type" : "VARCHAR(20)",
          "fromRoot": true } ] ' )
  ) AS JT(id, schoolName, "type", studentName);
```

```
*** Query completed. 4 rows found. 4 columns returned.
*** Total elapsed time was 1 second.
```

id	schoolName	type	studentName
1	Lake	elementary	Cameron
1	Madison	middle	Cameron
1	Rancho	high	Cameron
1	UCI	college	Cameron

Example: Using Extra Columns

To make the example simple, constants are used when possible from a column.

```
SELECT * FROM DATASET_Table (
  ON (SELECT id, DatasetCol, 'CA' AS state, 'USA' AS nation
      FROM my_table WHERE id=1)
  USING rowexpr('$schools[*]')
  colexpr(
```

```

        '[ {"dotnotation" : "$.name",
            "type" : "CHAR(20)"},
          {"dotnotation" : "$.type",
            "type" : "VARCHAR(20)"} ]')
) AS JT(id, name, "type", State, Nation);

*** Query completed. 4 rows found. 5 columns returned.
*** Total elapsed time was 1 second.

```

id	name	type	State	Nation
1	Lake	elementary	CA	USA
1	Madison	middle	CA	USA
1	Rancho	high	CA	USA
1	UCI	college	CA	USA

Example: Shredding a DATASET Values Table with the Storage Format CSV

This example shreds a table of DATASET values with the CSV storage format. The row in my_table with the ID 1 has the CSV value:

```

Item ID,Item Name,Quantity,Price
10021,basketball,10,29.99
10029,jersey,20,15.99
10032,shoes,20,79.99

SELECT * FROM DATASET_TABLE (
  ON (SELECT id, csvCol FROM my_table WHERE id=1)
  USING rowexpr('')
  colexpr(
    '[ {"dotnotation" : "$..Item Name",
        "type" : "CHAR(40)"},
      {"dotnotation" : "$..Price",
        "type" : "DECIMAL(9,2)"} ]')
  ) AS t1(id, Item_Name, Price);

```

The resulting table has three rows:

ID	Item_name	Price
1	basketball	29.99

ID	Item_name	Price
1	jersey	15.99
1	shoes	79.99

Examples: Shredding the CSV Data

You can import existing CSV data, represented as a VARCHAR or CLOB, to an existing relational table in the database. This is called "shredding" and uses the DATASET_TABLE table operator.

Example: Shredding the CSV Data to a Database Table

Convert the CSV data to a DATASET value with the CSV storage format by using CreateDATASET. Specify a schema if the CSV data does not follow the standard CSV format (for example, it has different column or record delimiters). The DATASET_TABLE then shreds the CSV data to a database table. Use DATASET_TABLE to specify each column data type in the CSV value by using the *'column_expression_literal* USING specification.

```
CREATE TABLE empTableFromCSV(
  empID INTEGER,
  empName VARCHAR(20),
  empSalary INTEGER);
```

Example: Aligning the CSV Columns and Table Columns

```
*+csv11.txt+*
empID,empName,empSalary
1,John,100000
2,Sam,50000
3,Mary,75000

.import vartext file csv11.txt
USING (c1 VARCHAR(1000))
INSERT INTO empTableFromCSV
SELECT * FROM DATASET_TABLE (
  ON (SELECT CreateDATASET(NULL, :c1, CSV))
  USING rowexpr('')
  colexpr(
    '[ {"dotnotation" : "$..empID",
      "type" : "INTEGER"},
    {"dotnotation" : "$..empName",
      "type" : "VARCHAR(20)"} ,
    {"dotnotation" : "$..empSalary",
```

```

        "type" : "INTEGER"} ]')
) AS Emp(empID, empName, empSalary);

SELECT * FROM empTableFromCSV ORDER BY 1;

```

The resulting table has three rows:

empID	empName	empSalary
1	John	100000
2	Sam	50000
3	Mary	75000

Example: CSV Column and Table Names are the Same, but Do Not Align

```

*+csv12.txt+*
empID,empSalary,empName
4,100000,Kate
5,50000,Rob
6,75000,Peter

```

To align the CSV column and table names:

```

.import vartext file csv12.txt
USING (c1 VARCHAR(1000))
INSERT INTO empTableFromCSV(empID,empSalary,empName)
SELECT * FROM DATASET_TABLE (
  ON (SELECT CreateDATASET(NULL, :c1, CSV))
  USING rowexpr('')
    colexpr(
      '[ {"dotnotation" : "$..empID",
        "type" : "INTEGER"},
        {"dotnotation" : "$..empSalary",
          "type" : "INTEGER"} ,
        {"dotnotation" : "$..empName",
          "type" : "VARCHAR(20)"} ]')
    ) AS Emp(empID, empSalary, empName);

```

Example: Passing the Schema to CreateDATASET

In this example, the CSV has no header, but the values align perfectly with the target table. You can pass the schema to the CreateDATASET function, indicating the CSV value has no header line.

```

*+csv13.txt+*
7,Matt,100000
8,Mark,50000
9,Luke,75000

.import vartext file csv13.txt
USING (c1 VARCHAR(1000))
INSERT INTO empTableFromCSV
SELECT * FROM DATASET_TABLE (
  ON (SELECT CreateDATASET('{ "field_names":NULL}':c1, CSV))
  USING rowexpr('')
  colexpr(
    '[ { "dotnotation" : "$..empID",
      "type" : "INTEGER"},
      { "dotnotation" : "$..empName",
        "type" : "VARCHAR(20)" } ,
      { "dotnotation" : "$..empSalary",
        "type" : "INTEGER" } ]')
  ) AS Emp(empID, empName, empSalary);

```

Example: Returning All Employees in the Table

```
SELECT * FROM empTableFromCSV ORDER BY 1;
```

empID	empName	empSalary
1	John	100000
2	Sam	50000
3	Mary	75000
4	Kate	100000
5	Rob	50000
6	Peter	75000
7	Matt	100000
8	Mark	50000
9	Luke	75000

DataSize

Returns the data length in bytes of any of the following Teradata variable maximum length complex data types:

- DATASET
- JSON
- ST_Geometry
- XML

Returns a BIGINT that is the size in bytes of the data object passed in to the function.

DataSize Syntax

```
[TD_SYSFNLIB.] DataSize (var_max_length_cdt)
```

Syntax Elements

var_max_length_cdt

A DATASET, JSON, ST_Geometry, or XML data type object.

DataSize Examples

The following examples use JSON data types.

```
SELECT TD_SYSFNLIB.DataSize (NEW JSON ('{"name" : "Mitzy", "age" : 3}'));
```

```
datasize( NEW JSON('{"name" : "Mitzy", "age" : 3}', LATIN))
-----
29
```

```
CREATE TABLE JSON_table (id INTEGER, jsn JSON INLINE LENGTH 1000);
```

```
INSERT INTO JSON_table VALUES (100, '{"name" : "Mitzy", "age" : 3}');
```

```
INSERT INTO JSON_table VALUES (200, '{"name" : "Rover", "age" : 5}');
```

```
INSERT INTO JSON_table VALUES (300, '{"name" : "Princess", "age" : 4.5}');
```

```
SELECT * FROM JSON_table ORDER BY id;
```

```
id jsn
```

```
-----
100 {"name" : "Mitzy", "age" : 3}
200 {"name" : "Rover", "age" : 5}
300 {"name" : "Princess", "age" : 4.5}
```

```
SELECT id, TD_SYSFNLIB.DataSize (jsn) FROM JSON_table ORDER BY id;
```

id	datasize(jsn)
-----	-----
100	29
200	29
300	34

SchemaEqual

SchemaEqual compares CHAR/VARCHAR/CLOB/JSON types representing a self-describing schema for equality. This function only compares Avro schemas.

SchemaEqual Syntax

```
SchemaEqual ( schema_expression, schema_expression )
```

Syntax Elements

schema_expression

Any expression that evaluates to a Teradata CHAR/VARCHAR/CLOB/JSON conforming to the schema specifications.

Note:

The following rules are for the Avro schema:

- The two values passed into this function must be valid Avro schemas.
- The SchemaEqual function imposes a more strict set of rules than SchemaMatch. To be equal, one of the following must occur:
 - Both schemas are arrays whose item types are equivalent (recursively determined if not primitive types).
 - Both schemas are maps whose value types are equivalent (recursively determined if not primitive types).
 - Both schemas are enums whose names or symbols match.
 - Both schemas are fixed with sizes and names that match.
 - Both schemas are records with the same name.

Fields of both records must be declared in the same order, have the same names, and the same data types (recursively determined if not primitive types).

- Both schemas are unions whose possible schema types are equivalent (recursively determined if not primitive types).
- Both schemas have the same primitive type.

Note:

A schema's "doc" fields are ignored for the purposes of schema equivalence evaluation.

Example: Comparing Data Types

This example compares CHAR/VARCHAR/CLOB/JSON data types that represent an Avro schema.

```
/*fail due to name of symbol mismatch*/
SELECT SchemaEqual(
'{ "type": "enum",
  "name": "Suit",
  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
}',
'{ "type": "enum",
  "name": "Suits",
  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
}');
> 0
```

```
/*fail due to structural mismatch*/
SELECT SchemaEqual(
'{ "type": "enum",
  "name": "Suit",
  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
}',
'{ "type" : "array",
  "items" : {
    "type": "record",
    "name": "test",
    "doc": "this is unique01",
    "fields" : [
      {"name": "a", "type": "long"},
      {"name": "b", "type": "string"}
    ]
  }
}');
> 0
```

```

/*success - everything is equal and doc is ignored*/
SELECT SchemaEqual(
  '{ "type" : "array",
    "items" : {
      "type": "record",
      "name": "test",
      "doc": "this is unique01",
      "fields" : [
        {"name": "a", "type": "long"},
        {"name": "b", "type": "string"}
      ]
    }
  }',
  '{ "type" : "array",
    "items" : {
      "type": "record",
      "name": "test",
      "doc": "this is unique02",
      "fields" : [
        {"name": "a", "type": "long"},
        {"name": "b", "type": "string"}
      ]
    }
  }');
> 1

```

SchemaMatch

SchemaMatch compares CHAR/VARCHAR/CLOB/JSON types used to represent a schema for compatibility. This function only compares Avro schemas.

SchemaMatch Syntax

```
SchemaMatch ( schema_expression, schema_expression )
```

Syntax Elements

schema_expression

Any expression that evaluates to a Teradata CHAR/VARCHAR/CLOB/JSON conforming to the schema specifications.

The two values passed into this function must be a valid schema.

Note:

The following information is from the Apache Avro specification. It is not Teradata specific; Teradata is implementing Apache's definition of matching schemas.

In this function, the first value passed in is regarded as the *writer's* schema and the second value passed in as the *reader's* schema. Both must represent a valid Avro schema, and the values must be a CHAR/VARCHAR/CLOB/JSON type.

To match, one of the following must occur:

- Both schemas are arrays whose item types match.
- Both schemas are maps whose value types match.
- Both schemas are enums whose names match.
- Both schemas are fixed whose sizes and names match.
- Both schemas are records with the same name.
- Either schema is a union.
- Both schemas have the same primitive type.
- The writer's schema may be promoted to the reader's schema if:
 - INT can be promoted to LONG, FLOAT, or DOUBLE.
 - LONG can be promoted to FLOAT or DOUBLE.
 - FLOAT can be promoted to DOUBLE.
 - The ordering of fields may be different. Fields are matched by name.
 - Schemas for fields with the same name in both records are resolved recursively.
 - If the writer's record contains a field with a name not present in the reader's record, the writer's value for that field is ignored.
 - If the reader's record schema has a field that contains a default value, and the writer's schema does not have a field with the same name, then use the default value from the reader's schema field.
 - If the reader's record schema has a field with no default value, and the writer's schema does not have a field with the same name, the schemas do not match.
- If both are enums and the writer's symbol is not present in the reader's enum, then the schemas do not match.
- If both are arrays: The resolution algorithm is applied recursively to the reader's and writer's array item schemas.
- If both are maps: The resolution algorithm is applied recursively to the reader's and writer's value schemas.
- If both are unions: The first schema in the reader's union that matches the selected writer's union schema is recursively resolved against it. If none match, the schemas do not match.

- If the reader's schema is a union, but writer's is not, then the first schema in the reader's union that matches the writer's schema is recursively resolved against it. If none match, the schemas do not match.
- If the writer's schema is a union, but the reader's schema is not:
 - If the reader's schema matches the selected writer's schema, it is recursively resolved against it.
 - If they do not match, the schemas do not match.
- A schema's "doc" fields are ignored for the purposes of schema resolution.

Example: Matching Schemas

The following schemas present various examples of schema matching. Some have matching schemas, while others do not.

```
/*fail due to name of symbol mismatch*/
SELECT SchemaMatch(
  '{ "type": "enum",
    "name": "Suit",
    "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
  }'
,
  '{ "type": "enum",
    "name": "Suits",
    "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
  }');
> 0

/*fail due to structural mismatch*/
SELECT SchemaMatch(
  '{ "type": "enum",
    "name": "Suit",
    "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
  }'
,
  '{ "type" : "array",
    "items" : {
      "type": "record",
      "name": "test",
      "doc": "this is unique01",
      "fields" : [
        {"name": "a", "type": "long"},
        {"name": "b", "type": "string"}
      ]
    }
  }');
```

```

    ]
  }}');
> 0

/*success - doc is ignored, ordering of names does not matter, long promotes to
double, and int promotes to float*/
SELECT SchemaMatch(
'{ "type" : "array",
  "items" : {
    "type": "record",
    "name": "test",
    "doc": "this is unique01",
    "fields" : [
      {"name": "a", "type": "long"},
      {"name": "b", "type": "int"}
    ]
  }},
'{ "type" : "array",
  "items" : {
    "type": "record",
    "name": "test",
    "doc": "this is unique02",
    "fields" : [
      {"name": "b", "type": "float"},
      {"name": "a", "type": "double"}
    ]
  }},
  }}');
> 1

```

CSV Conversion

With the system-defined CSV conversion feature, you can convert data in the CSV format to a JSON or DATASET data type with storage format Avro or CSV. This section describes the various table operators that allow for conversion.

Examples in the section use the following table several times:

```
CREATE TABLE myCSVTable09(
    id INTEGER,
    csvFile CLOB);

csv09.txt (note that the record delimiter is shown as '\n')
Item_ID,Item_Name,Item_Color,Item_Style,Quantity_Purchased,Item_Price,Total_Price
\n55,bicycle,red,boys,1,100.00,100.00 \n88,toy boat,pink,,1,15.10,15.10
\n105,soap,,,1,0.99,0.99|1

.import vartext file csv09.txt
USING (c1 VARCHAR(1000), c2 INTEGER)
INSERT INTO myCSVTable09(:c2,:c1);
```

The CSV data is included for reference:

Item_ID	Item_Name	Item_Color	Item_Style	Quantity_Purchased	Item_Price	Total_Price
55	bicycle	red	boys	1	100.00	100.00
88	toy boat	pink	?	1	15.10	15.10
105	soap	?	?	1	0.99	0.99

CSV Schema

The DATASET type supports the CSV storage format.

A simple schema is required for specifying certain optional attributes of CSV. Specify the schema as a JSON document composed of specific key-value pairs.

Example: Using a CSV Schema

The following example contains attributes based on some optional parameters.


```
{
  "field_names" : <JSON array with column names >,
  "field_delimiter" : <field_delimiter_character>,
  "record_delimiter" : <record_delimiter_character>
}
```

Another example of the schema:

```
{
  "field_names" : ["name","dob","phone","address"],
  "field_delimiter" : ",",
  "record_delimiter" : "\r\n"
}
```

Note that the key names `field_names`, `field_delimiter`, and `record_delimiter` must be specified exactly as shown to be correctly interpreted. The names are also case-sensitive.

There are three options when specifying a header for CSV data using this schema format:

1. The `field_names` key/value pair is omitted. This tells the database there is a header record in the CSV data.
2. The `field_names` key/value pair is included, but its value is a JSON null. This tells the database that there is no header included in the CSV data, and that you do not want to provide one. In this case, the database auto-generates names for the fields of the CSV file, according to the format:

```
csv_fld1, csv_fld2, csv_fld3, ... , csv_fldN
```

3. The `field_names` key/value pair is included, and its value is a JSON array of strings. This tells the database there is no header in the CSV data, and that you want to specify the names of the fields by using this schema.

Based on the schema, the data is expected to be a set of characters where fields are divided by the `field_delimiter` and records are divided by the `record_delimiter`. The following example shows CSV data which has `'&'` as the field delimiter, `'#'` as the record delimiter, and contains a header row:

Schema

```
{
  "field_delimiter" : "&",
  "record_delimiter" : "#"
}
```

Data

```
Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total
Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
```

CSV_TO_AVRO

Use the CSV_TO_AVRO table operator to convert CSV data into Avro data.

CSV_TO_AVRO Syntax

```
CSV_TO_AVRO ( ON
  ( SELECT CSV_expression select_stmt_options )
  [ USING { SCHEMA ( schema ) | AGGREGATE ( {'Y'|'N'} ) } ]
)
```

Syntax Elements

CSV_expression

Any expression that evaluates to character data in the CSV format or a DATASET STORAGE FORMAT CSV value.

select_stmt_options

The *select_stmt_options* are the allowable or necessary options in Vantage for a SQL SELECT statement.

USING

Variables used as input to the SQL statement specified by *select_stmt_options*.

schema

The CSV schema.

AGGREGATE

Specify AGGREGATE for the output instances to be composed of one row of input data.

Usage Notes

Specifying Input Values

The first input value specified via the ON clause must be character based (for example, a CHAR/ VARCHAR/CLOB or a DATASET STORAGE FORMAT CSV value) and be composed of data that

conforms to the specified format (that is, it uses the specified field/record delimiters, or defaults if not specified). Any subsequent input values do not affect the result of conversion to Avro or JSON, but are passed through the table operator and given as additional output columns. This allows the resulting Avro or JSON documents to be associated with the source CSV data, which is very important when converting multiple CSV inputs.

```
CT my_table(id int, csvData VARCHAR(500));
INSERT INTO my_table(1, 'a,b,c,d,e,f\1,2,3,4,5,6\7,8,9,10,11,12');
INSERT INTO my_table(2, 'a,b,c,d,e,f\13,14,15,16,17,18
\19,20,21,22,23,24');
INSERT INTO my_table(3, 'a,b,c,d,e,f\25,26,27,28,29,30
\31,32,33,34,35,36');
```

```
SELECT id, data.toJSON() FROM CSV_TO_AVRO
(
  ON (SELECT csvData, id FROM my_table)
  USING SCHEMA('{ "record_delimiter": "\\\n" }')
) as csvAvro
ORDER BY id, data.a;
```

```
id data.TOJSON()
```

```
-----
1 {"a": "1", "b": "2", "c": "3", "d": "4", "e": "5", "f": "6"}
1 {"a": "7", "b": "8", "c": "9", "d": "10", "e": "11", "f": "12"}
2 {"a": "13", "b": "14", "c": "15", "d": "16", "e": "17", "f": "18"}
2 {"a": "19", "b": "20", "c": "21", "d": "22", "e": "23", "f": "24"}
3 {"a": "25", "b": "26", "c": "27", "d": "28", "e": "29", "f": "30"}
3 {"a": "31", "b": "32", "c": "33", "d": "34", "e": "35", "f": "36"}
```

Rules and Restrictions

CSV_TO_AVRO converts CSV format data into DATASET STORAGE FORMAT AVRO or CSV data type instances. CSV_TO_AVRO produces one output column called 'data'.

The RETURNS clause defines the resulting Avro instance variable inline and maximum length. The default is to return a maximum size DATASET STORAGE FORMAT AVRO or CSV instance. All values are treated as strings, except the NULL fields are converted to null Avro or CSV values.

For more information about Rules and Restrictions, see [CSV_TO_JSON](#).

Converting CSV to Avro

The following example converts CSV to Avro, where each CSV record is converted to one output row composed of one Avro record, along with its schema.

```
csv10.txt
Item_ID,Item_Name,Item_Color,Item_Style,Quantity_Purchased,Item_Price,Total_Pric
e#55,bicycle,red,boys,1,100.00,100.00#88,toy
boat,pink,,1,15.10,15.10#105,soap,,,1,0.99,0.99|2

CREATE TABLE myCSVTable09(
    id INTEGER,
    csvFile CLOB);

.import vartext file csv10.txt
USING (c1 VARCHAR(1000), c2 VARCHAR(10))
INSERT INTO myCSVTable09(cast(:c2 AS INTEGER),:c1);
```

Examples

Example: Using the SCHEMA Custom Clause to Specify Non-Standard Data

If there is non-standard CSV data, use the SCHEMA custom clause to specify non-standard data.

```
SELECT data.toJSON() FROM CSV_TO_AVRO
(
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
    USING SCHEMA('{"record_delimiter":"#"}')
) AS csvAvro ORDER BY data."Item_ID";

{"Item_ID":"105","Item_Name":"soap","Item_Color":null,"Item_Style":null,"Quantit
y_Purchased":"1","Item_Price":"0.99","Total_Price":"0.99"}
{"Item_ID":"55","Item_Name":"bicycle","Item_Color":"red","Item_Style":"boys","Qu
antity_Purchased":"1","Item_Price":"100.00","Total_Price":"100.00"}
{"Item_ID":"88","Item_Name":"toy
boat","Item_Color":"pink","Item_Style":null,"Quantity_Purchased":"1","Item_Price
":"15.10","Total_Price":"15.10"}
```

Example: Changing the Key Names in the Avro Record

To change the names of the keys in the Avro records, use the SCHEMA custom clause.

```
SELECT data.toJSON() FROM CSV_TO_AVRO
(
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
    USING SCHEMA('{ "record_delimiter": "#", "field_names":
["a1", "a2", "a3", "a4", "a5", "a6", "a7"]}')
) AS csvAvro ORDER BY data."a1";

> {"a1": "105", "a2": "soap", "a3": null, "a4": null, "a5": "1", "a6": "0.99", "a7": "0.99"}
>
{"a1": "55", "a2": "bicycle", "a3": "red", "a4": "boys", "a5": "1", "a6": "100.00", "a7": "10
0.00"}
> {"a1": "88", "a2": "toy
boat", "a3": "pink", "a4": null, "a5": "1", "a6": "15.10", "a7": "15.10"}
>
{"a1": "Item_ID", "a2": "Item_Name", "a3": "Item_Color", "a4": "Item_Style", "a5": "Quant
ity_Purchased", "a6": "Item_Price", "a7": "Total_Price"}
```

Example: Using CSV Data in Double Quotes

Some CSV data fields are wrapped in double quotes, which is in line with the CSV specification at <https://tools.ietf.org/html/rfc4180#section-2>. A field or key is allowed to have a delimiter in it when surrounded by double quotes denoting the delimiter is part of the value.

Example: Aggregating Avro Output

Aggregate output into one Avro array composed of Avro records mapping to each record of the CSV data.

```
SELECT data.toJSON() FROM CSV_TO_AVRO
(
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
    USING SCHEMA('{ "record_delimiter": "#"}') DOAggregate('Y')
) AS csvAvro;

> [{"Item_ID": "55", "Item_Name": "bicycle", "Item_Color": "red",
"Item_Style": "boys", "Quantity_Purchased": "1", "Item_Price": "100.00",
"Total_Price": "100.00"}, {"Item_ID": "88", "Item_Name": "toy boat",
"Item_Color": "pink", "Item_Style": null, "Quantity_Purchased": "1",
```

```
"Item_Price":"15.10", "Total_Price":"15.10"}, {"Item_ID":"105",
"Item_Name":"soap", "Item_Color":null, "Item_Style":null,
"Quantity_Purchased":"1", "Item_Price":"0.99", "Total_Price":"0.99"}]
```

Example: Converting a DATASET STORAGE FORMAT CSV Value to AVRO

In this example, each CSV record is converted to one output row composed of one AVRO record, along with its schema.

```
CREATE TABLE csv_table_1(pkey INTEGER,
csv DATASET STORAGE FORMAT CSV);
INSERT INTO csv_table_1 values(0,
'ItemID,ItemName,Quantity,Price;10021,Paint
Brush,10,10.99;10033,Paint,3,24.99');
SELECT data.ToJson() FROM CSV_TO_AVRO
(
ON (SELECT csv FROM csv_table_1)
USING SCHEMA('{"record_delimiter":";"}')
) AS csv;
> {"ItemID":"10021","ItemName":"Paint Brush","Quantity":"10","Price":
"10.99"}
> {"ItemID":"10033","ItemName":"Paint","Quantity":"3","Price":"24.99"}
```

CSV_TO_JSON

The CSV_TO_JSON table operator converts CSV data into JSON data.

CSV_TO_JSON Syntax

```
CSV_TO_JSON ( ON
( SELECT CSV_expression select_stmt_options )
[ USING { SCHEMA ( schema ) | AGGREGATE ( { 'Y' | 'N' } ) } ]
)
```

Syntax Elements

CSV_expression

Any expression that evaluates to character data in the CSV format or a DATASET STORAGE FORMAT CSV value.

select_stmt_options

The *select_stmt_options* are the allowable or necessary options in Vantage for a SQL SELECT statement.

USING

Variables used as input to the SQL statement specified by *select_stmt_options*.

schema

The CSV schema.

AGGREGATE

Specify AGGREGATE for the output instances to be composed of one row of input data.

Usage Notes

Specifying Input Values

The first input value specified via the ON clause must be character based (for example, a CHAR/VARCHAR/CLOB or a DATASET STORAGE FORMAT CSV value), and it must be composed of data that conforms to the specified format. That is, it uses the specified field/record delimiters, or defaults if not specified. Any subsequent input values do not affect the result of conversion to Avro or JSON, but are passed through the table operator and given as additional output columns. This allows the resulting Avro or JSON documents to be associated with the source CSV data, which is very important when converting multiple CSV inputs.

```
CT dr181746_table(id int, csvData VARCHAR(500));
INSERT INTO dr181746_table(1, 'a,b,c,d,e,f\1,2,3,4,5,6\7,8,9,10,11,12');
INSERT INTO dr181746_table(2, 'a,b,c,d,e,f\13,14,15,16,17,18
\19,20,21,22,23,24');
INSERT INTO dr181746_table(3, 'a,b,c,d,e,f\25,26,27,28,29,30
\31,32,33,34,35,36');

SELECT id, data FROM CSV_TO_JSON
(
  ON (SELECT csvData, id FROM dr181746_table)
  USING SCHEMA('{ "record_delimiter": "\\\\"}')
) as csvJSON
ORDER BY id, data.a;

      id data.TOJSON()
-----
```

```

1 {"a": "1", "b": "2", "c": "3", "d": "4", "e": "5", "f": "6"}
1 {"a": "7", "b": "8", "c": "9", "d": "10", "e": "11", "f": "12"}
2 {"a": "13", "b": "14", "c": "15", "d": "16", "e": "17", "f": "18"}
2 {"a": "19", "b": "20", "c": "21", "d": "22", "e": "23", "f": "24"}
3 {"a": "25", "b": "26", "c": "27", "d": "28", "e": "29", "f": "30"}
3 {"a": "31", "b": "32", "c": "33", "d": "34", "e": "35", "f": "36"}

```

Rules and Restrictions

When CSV_TO_JSON converts CSV data into JSON data type instances, the input consists of multiple sets of CSV data with these expected behaviors:

- If the data is aggregated, the structure must be identical.
- If the data is not aggregated, the structure does not have to be identical. However, if the SCHEMA custom clause specifies field names in the output JSON documents, the numbers of specified names and fields in every record of every CSV data set must be equal.

CSV_TO_JSON produces one output column called 'data'. You can tune the output based on the following custom clauses:

- Specify SCHEMA to explicitly define the published data structure. Use the clause with a character string representing an ad-hoc schema specification; any other value results in an error. If an ad-hoc schema is specified, the structure must conform to the CSV format rules. If the clause is not specified, the input CSV data is assumed to conform to the defaults.
- Specify DO_AGGREGATE output instances to compose input data as one row. The clause accepts either Y (the result is aggregated) or N (the result is not aggregated, which is the default). Neither option is case-sensitive. If the clause is excluded, the table operator returns one JSON instance in one Teradata output row for each record for each set of input CSV data. If the aggregated data results in a size overflow based on the maximum size specified, an error occurs.

The table operator uses the RETURNS clause to compose data into a JSON data type of any character set or storage format. The default is to return maximum size JSON CHARACTER SET LATIN instances. All values are treated as strings, except NULL fields, which are converted to null JSON values.

Fields in CSV data may be wrapped in double quotes, especially when the field contains characters used as a field or record delimiter. When CSV_TO_JSON encounters a double-quoted field, it outputs the key or value without the extra quotes. See *Example: Using CSV Data in Double Quotes (CSV to JSON)* to view a record delimiter containing the CSV data field name, and to see that the extra quotes are removed before constructing the JSON document.

Note that commas at the end of a record with no data (except a record delimiter) following them are ignored. The commas are not interpreted as a null value.

Examples

Example: Converting CSV to JSON

The following is a simple example of converting CSV to JSON, where each CSV record is converted to one output row composed of one JSON object.

```
SELECT * FROM CSV_TO_JSON
(
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
    USING SCHEMA('{"record_delimiter":"#"}')
) AS csvJSON ORDER BY data."Item ID";

> {"Item_ID":"55", "Item_Name":"bicycle", "Item_Color":"red",
  "Item_Style":"boys", "Quantity_Purchased":"1", "Item_Price":"100.00",
  "Total_Price":"100.00"}
> {"Item_ID":"88", "Item_Name":"toy boat", "Item_Color":"pink",
  "Item_Style":null, "Quantity_Purchased":"1", "Item_Price":"15.10",
  "Total_Price":"15.10"}
> {"Item_ID":"105", "Item_Name":"soap", "Item_Color":null, "Item_Style":null,
  "Quantity_Purchased":"1", "Item_Price":"0.99", "Total_Price":"0.99"}
```

Example: Changing the Key Names in the Avro Record

To change the key names in the resulting JSON documents, use the SCHEMA custom clause.

```
SELECT * FROM CSV_TO_JSON
(
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
    USING SCHEMA('{"record_delimiter":"#", "field_names":
  ["a1","a2","a3","a4","a5","a6","a7"]}')
```

```
) AS csvJSON ORDER BY data."a1";

> {"a1":"55", "a2":"bicycle", "a3":"red", "a4":"boys", "a5":"1", "a6":"100.00",
  "a7":"100.00"}
> {"a1":"88", "a2":"toy boat", "a3":"pink", "a4":null, "a5":"1", "a6":"15.10",
  "a7":"15.10"}
> {"a1":"105", "a2":"soap", "a3":null, "a4":null, "a5":"1", "a6":"0.99",
  "a7":"0.99"}
```

Example: Using CSV Data in Double Quotes

Some CSV data feature fields are wrapped in double quotes. The double quotes allow invalid characters (such as record/field delimiters or double quotes) in the value itself. In the following example, CSV_TO_JSON omits the extra leading and trailing double quotes from the output.

```
SELECT * FROM CSV_TO_JSON
(
    ON (SELECT
        'Item_ID,Item_Name,"Item#Color",Item_Style,Quantity_Purchased,Item_Price,Total_P
rice#55,bicycle,red,boys,1,100.00,100.00#88,toy
boat,pink,,1,15.10,15.10#105,soap,,,1,0.99,0.99')
        USING SCHEMA({'record_delimiter':"#"})
    ) AS csvJSON ORDER BY data."Item_ID";

> {"Item_ID":"55", "Item_Name":"bicycle", "Item#Color":"red",
  "Item_Style":"boys", "Quantity_Purchased":"1", "Item_Price":"100.00",
  "Total_Price":"100.00"}
> {"Item_ID":"88", "Item_Name":"toy boat", "Item#Color":"pink",
  "Item_Style":null, "Quantity_Purchased":"1", "Item_Price":"15.10",
  "Total_Price":"15.10"}
> {"Item_ID":"105", "Item_Name":"soap", "Item#Color":null, "Item_Style":null,
  "Quantity_Purchased":"1", "Item_Price":"0.99", "Total_Price":"0.99"}
```

Example: Aggregating Avro Output

Output may be aggregated into one JSON array composed of JSON objects mapping to the CSV data.

```
SELECT * FROM CSV_TO_JSON
(
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
        USING SCHEMA({'record_delimiter':"#"}) DO_AGGREGATE('Y')
    ) AS csvJSON;

> [{"Item_ID":"55", "Item_Name":"bicycle", "Item_Color":"red",
  "Item_Style":"boys", "Quantity_Purchased":"1", "Item_Price":"100.00",
  "Total_Price":"100.00"}, {"Item_ID":"88", "Item_Name":"toy boat",
  "Item_Color":"pink", "Item_Style":null, "Quantity_Purchased":"1",
  "Item_Price":"15.10", "Total_Price":"15.10"}, {"Item_ID":"105",
  "Item_Name":"soap", "Item_Color":null, "Item_Style":null,
  "Quantity_Purchased":"1", "Item_Price":"0.99", "Total_Price":"0.99"}]
```

Example: Converting a DATASET STORAGE FORMAT CSV Value to JSON

In this example, each CSV record is converted to one output row composed of one JSON object.

```
CREATE TABLE csv_table_1(pkey INTEGER,  
csv DATASET STORAGE FORMAT CSV);  
INSERT INTO csv_table_1 values(0,  
'ItemID,ItemName,Quantity,Price;10021,Paint  
Brush,10,10.99;10033,Paint,3,24.99');  
SELECT data FROM CSV_TO_JSON  
(  
ON (SELECT csv FROM csv_table_1)  
USING SCHEMA('{ "record_delimiter": ";" }')  
) AS csv;  
> {"ItemID":"10021","ItemName":"Paint Brush","Quantity":"10","Price":  
"10.99"}  
> {"ItemID":"10033","ItemName":"Paint","Quantity":"3","Price":"24.99"}
```

DATASET Publishing

About Publishing

The database provides the DATASET publishing functionality to compose a DATASET document using various parameters. You can create simple or complex queries to generate the DATASET document.

DATASET_PUBLISH

The DATASET_PUBLISH table operator allows any row of data in a relational table to be composed into a complex data type.

Usage Notes

DATASET_PUBLISH composes a DATASET data type instance from different data sources (anything referenced in an SQL statement). It publishes DATASET data types of any storage format, exporting stored data stored to an externally recognizable file format. DATASET_PUBLISH returns one output column, "data," which returns the data composed as a result of this operation.

The formats vary:

- Data composed as Avro is returned as an instance of the DATASET type with storage format AVRO. When a DATASET data type stored as AVRO creates an Avro instance by using DATASET_PUBLISH, its schema is combined with the output schema. The default is to publish DATASET values in the AVRO storage format.
- Data composed as CSV returns as a DATASET type instance with storage format CSV. The CSV file contains a header line with CSV column names and data. If the specified schema has a null field_names key, no header is included with the CSV value. All data is separated using the row and column delimiters. All Teradata complex data types (including DATASET) creating a CSV instance with DATASET_PUBLISH are converted to text representation and treated as one field in the CSV instance. To include a schema in the DATASET type, use the SCHEMA clause.

If the DATASET value has a schema and is inserted into a table column with a column-based schema defined, the schema is removed from the value, and the CSV data is validated against the column-based schema.

JSON data and DATASET data are not supported as input.

- JSON data published to a DATASET type is converted to the desired storage format and then combined with the output.
- Any other Teradata complex data types used to create an AVRO or CSV instance by using DATASET_PUBLISH are converted to text representation and treated as one field in the resulting instance. Note that distinct and structured UDTs are not supported.

When DATASET_PUBLISH is used to compose an Avro instance and you are publishing types that are found as nullable by definition, the schema marks the type as such by using a union of null and the corresponding data type. This means that the schema type for a field will look like "type": ["null", "int"] instead of "type": "int".

Note:

Null is always guaranteed to be the first element in the union for the auto-generated Avro schema.

Using Custom Clauses with DATASET_PUBLISH

The DATASET_PUBLISH table operator uses custom clauses to compose DATASET data type instances in your chosen format:

- Specify SCHEMA to explicitly define the published data structure. Use SCHEMA with a character string representing an ad-hoc schema specification; any other value results in an error. When specifying an ad-hoc schema, the structure must conform to the output storage format rules. If this clause is not specified, the schema is automatically generated by the database based on the data used to compose the DATASET data type instances.
- Specify DO_AGGREGATE for output instances composed of one row of output data. The DO_AGGREGATE clause accepts either Y or N, where Y signifies that the result is aggregated (default) and N is not aggregated. The values are not case-sensitive. If DO_AGGREGATE is excluded, the table operator aggregates all data corresponding to a particular group (as defined in the optional GROUP BY clause in the SELECT statement of the ON clause) into one DATASET data type instance.
- Specify UNIQUE_NAMES to generate UNIQUE record or fixed type names when constructing the output schema. This is helpful when nesting DATASET or JSON data within the final result. The UNIQUE_NAMES clause accepts either Y or N (not case-sensitive) where Y signifies each auto-generated name is unique, and N is not unique. The default is N. "UNIQUE_NAMES" is ignored if the storage format is CSV.
- Specify INCLUDE_HEADER only for DATASET values in CSV storage format. INCLUDE_HEADER accepts either Y or N, (not case-sensitive) where Y signifies each returned DATASET value includes a header record as the first record in the file. N signifies omitting the header record. The default is Y, which includes the header record.

If SCHEMA supplies the CSV schema, with field_names specified and a non-null value, the DATASET value includes a header that uses the SCHEMA-supplied field names. The field names are then removed from the schema. If INCLUDE_HEADER is specified as N, then no header record is written; instead the field names are referenced from the schema.

If a CSV schema is not supplied by SCHEMA, and INCLUDE_HEADER is Y, then the table column names are written as the header record to the CSV file. If INCLUDE_HEADER is N, field_names is null in the schema.

If SCHEMA provides a CSV schema and field_names is null, with INCLUDE_HEADER as Y, an error occurs because null is an invalid header record. If INCLUDE_HEADER is N, no action is taken.

The following table summarizes all combinations of the SCHEMA and INCLUDE_HEADER clauses.

Schema	Include Header	Action
No Schema	Y	Use table column names as header record
	N	Create schema with "field_names":null
Schema with Field Names	Y	<ul style="list-style-type: none"> • Use field names as header • Remove field names from schema
	N	<ul style="list-style-type: none"> • No action • Refers to the schema for field names
Schema with "field_names":null OR "field_names":[]	Y	Error
	N	<ul style="list-style-type: none"> • No action • No header record or field names are assumed

Additionally, use the RETURNS clause to specify the output data type or length. The default is to publish to AVRO of maximum length, so omit RETURNS if for that desired output. For CSV storage format output, specify RETURNS. The return type must be a DATASET STORAGE FORMAT CSV data type.

If the result set is inserted into table, the schema is stored in every DATASET result value unless the DATASET table column has a schema associated with it from the WITH SCHEMA clause in the CREATE TABLE statement. In that case, the schema is not stored with every value.

Publishing NUMBER or DECIMAL Types to an Avro Value

When publishing NUMBER or DECIMAL types from a table to an Avro value, the schema is written out as a string instead of a double type. Because NUMBER and DECIMAL use a higher precision, using a double or other relevant primitive type can result in a loss of precision. Using a string prevents this loss of information.

The schema looks like the following when using NUMBER or DECIMAL types:

```
CREATE TABLE avrotest(a INTEGER, b DECIMAL(38,9), c NUMBER(38,9));
```

```
INSERT INTO avrotest(1, 12345678901234567890123456789.012345679,
12345678901234567890123456789.012345679);
```

```
SELECT data.tojson(), data.getschema() FROM DATASET_PUBLISH
(
```

```
ON (SELECT * FROM avrotest)
) as L;
```

```
*** Query completed. One row found. 2 columns returned.
```

```
*** Total elapsed time was 1 second.
```

```
data.TOJSON() [{"a":{"int":1},"b":
{"string":"12345678901234567890123456789.012345679"},"c":
{"string":"12345678901234567890123456789.012345679"}}]
data.GETSCHEMA()
{"type":"array","items":{"type":"record","name":"rec_0","fields":
[{"name":"a","type":["null","int"]},{ "name":"b","type":["null","string"]},
{"name":"c","type":["null","string"]}]}]}
```

Examples

Example: Composing a Table to a DATASET Data Type

This example shows how to compose an entire table to a DATASET data type.

```
CREATE TABLE employeeTable(
    empID INTEGER,
    empName VARCHAR(100),
    empDept VARCHAR(100));
INSERT INTO employeeTable(1,'George Smith','Accounting');
INSERT INTO employeeTable(2,'Pauline Kramer','HR');
INSERT INTO employeeTable(3,'Steven Mazzo','Engineering');

SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT empName, empDept FROM employeeTable)

) AS avroFiles;

data
7B2274797065223A226172726179222C226974656D73223A7B2274797065223A227265636F726422
2C226E616D65223A227265635F30222C226669656C6473223A5B7B226E616D65223A22656D704E61
6D65222C2274797065223A22737472696E67227D2C7B226E616D65223A22656D7044657074222C22
74797065223A22737472696E67227D5D7D7D0006001847656F72676520536D697468144163636F75
6E74696E67021C5061756C696E65204B72616D6572044852041853746576656E204D617A7A6F1645
6E67696E656572696E6700
```

To compose an entire table to a DATASET data type, with the storage format CSV:

```
CREATE TABLE employeeTable(
    empID INTEGER,
    empName VARCHAR(100),
    empDept VARCHAR(100));
INSERT INTO employeeTable(1,'George Smith','Accounting');
INSERT INTO employeeTable(2,'Pauline Kramer','HR');
INSERT INTO employeeTable(3,'Steven Mazzo','Engineering');

SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT empName, empDept FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT CSV)

) AS csvFiles;
```

fileSchema	fileData
Null	empName,empDept George Smith,Accounting Pauline Kramer,HR Steven Mazzo,Engineering

To convert the output to JSON, run the following examples using 'SELECT data.toJSON()' instead of 'SELECT *'.

Example: Specifying Schemas

In this example, specify schemas for the output format AVRO:

```
SELECT data.ToJson() FROM DATASET_PUBLISH
(
    ON (SELECT * FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT AVRO)
    USING SCHEMA
    ( '
{
    "type": "array",
    "items": {
        "type": "record",
        "name": "employeeRecord",
```



```

        "fields": [
        {
            "name": "empID",
            "type": ["null","int"]
        },
        {
            "name": "empName",
            "type": ["null","string"]
        },
        {
            "name": "empDept",
            "type": ["null","string"]
        }
        ]
    }
    ')
) AS avroFiles;

data.ToJson()
[{"empID":{"int":3},"empName":{"string":"Steven Mazzo"},"empDept":
{"string":"Engineering"}},{ "empID":{"int":1},"empName":{"string":"George
Smith"},"empDept":{"string":"Accounting"}},{ "empID":{"int":2},"empName":
{"string":"Pauline Kramer"},"empDept":{"string":"HR"}}]

```

Or specify the schema for the output format CSV:

```

SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT * FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT CSV)
    USING SCHEMA
    ('{"field_delimiter": "*" , "record_delimiter" : "/" ,
    "field_names" : ["empIdentifier", "empFullName",
    "department"] }')
) AS csvFiles;

```

The following are the results for the table:

fileSchema	fileData
<pre>{ "field_delimiter": "*", "record_delimiter": "/" }</pre>	<pre>empIdentifier*empFullName*department/1*George Smith*Accounting/2*Pauline Kramer*HR/3*Steven Mazzo*Engineering</pre>

fileSchema	fileData
}	

Example: Publishing Table Columns

In this example using DATASET STORAGE FORMAT Avro, not all of the table columns are published.

```
SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT empName, empDept FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT Avro)
) AS avroFiles;

data
7B2274797065223A226172726179222C226974656D73223A7B2274797065223A227265636F726422
2C226E616D65223A227265635F30222C226669656C6473223A5B7B226E616D65223A22656D704E61
6D65222C2274797065223A22737472696E67227D2C7B226E616D65223A22656D7044657074222C22
74797065223A22737472696E67227D5D7D7D0006001847656F72676520536D697468144163636F75
6E74696E67021C5061756C696E65204B72616D6572044852041853746576656E204D617A7A6F1645
6E67696E656572696E6700
```

This example uses DATASET STORAGE FORMAT CSV:

```
SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT empName, empDept FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT CSV)
) AS csvFiles;
```

fileSchema	fileData
Null	empName,empDept George Smith,Accounting Pauline Kramer,HR Steven Mazzo,Engineering

Example: Returning Results that are Not Aggregated

In the following AVRO example, the results are not aggregated. For example, each row of a table results in one DATASET data type instance.

```

SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT * FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT AVRO)
    USING DO_AGGREGATE('N')
) AS avroFiles;

data
7B226E616D65223A22656D706C6F7965655461626C65222C2274797065223A227265636F7264222C
226669656C6473223A5B7B226E616D65223A22656D704944222C2274797065223A22696E74227D2C
7B226E616D65223A22656D704E616D65222C2274797065223A22737472696E67227D2C7B226E616D
65223A22656D7044657074222C2274797065223A22737472696E67227D5D7D00021847656F726765
20536D697468144163636F756E74696E67

7B226E616D65223A22656D706C6F7965655461626C65222C2274797065223A227265636F7264222C
226669656C6473223A5B7B226E616D65223A22656D704944222C2274797065223A22696E74227D2C
7B226E616D65223A22656D704E616D65222C2274797065223A22737472696E67227D2C7B226E616D
65223A22656D7044657074222C2274797065223A22737472696E67227D5D7D00041C5061756C696E
65204B72616D6572044852

7B226E616D65223A22656D706C6F7965655461626C65222C2274797065223A227265636F7264222C
226669656C6473223A5B7B226E616D65223A22656D704944222C2274797065223A22696E74227D2C
7B226E616D65223A22656D704E616D65222C2274797065223A22737472696E67227D2C7B226E616D
65223A22656D7044657074222C2274797065223A22737472696E67227D5D7D00061853746576656E
204D617A7A6F16456E67696E656572696E67

```

The results are not aggregated in the CSV example:

```

SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT * FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT CSV)
    USING DO_AGGREGATE('N')
) AS csvFiles;

```

fileSchema	fileData
Null	empld,empname,empDept 1,George Smith,Accounting
Null	empld,empname,empDept 2,Pauline Kramer,HR
Null	empld,empname,empDept

fileSchema	fileData
	3,Steven Mazzo,Engineering

Example: Using the UNIQUE_NAMES Clause

The schema output is different (displayed via the getSchema method of the DATASET type).

```
SELECT data.getSchema() FROM DATASET_PUBLISH
(
  ON (SELECT * FROM employeeTable)
  USING UNIQUE_NAMES('Y')
) AS avroFiles;
```

```
data.getSchema()
{
  "type": "array",
  "items": {
    "type": "record",
    "name": "rec_0_1448383894",
    "fields": [{
      "name": "empID",
      "type": "int"
    },
    {
      "name": "empName",
      "type": "string"
    },
    {
      "name": "empDept",
      "type": "string"
    }
  ]
}
```

Example: Aggregating Multiple Values from Multiple Rows Into One Instance

Use DATASET_PUBLISH to aggregate multiple values from multiple rows into one instance locally on each AMP. This AMP-local aggregation is standard operating procedure for table operators.

DATASET_PUBLISH may be invoked twice within a statement to perform a single aggregation of all input values. If DATASET_PUBLISH is invoked once without the PARTITION BY clause, each AMP produces one row of aggregated output. Some input data is added to the source table in this example.

```
INSERT INTO employeeTable(4,'Jose Hernandez','Engineering');
INSERT INTO employeeTable(5,'Kyle Newman','Engineering');
INSERT INTO employeeTable(6,'Pamela Giles','Sales');
```

```
SELECT data.toJSON() FROM DATASET_PUBLISH
(
    ON (SELECT * FROM employeeTable)
) AS avroFiles;
```

```
data.toJSON()
[
  {
    "empID": 5,
    "empName": "Kyle Newman",
    "empDept": "Engineering"
  },
  {
    "empID": 3,
    "empName": "Steven Mazzo",
    "empDept": "Engineering"
  },
  {
    "empID": 1,
    "empName": "George Smith",
    "empDept": "Accounting"
  },
  {
    "empID": 2,
    "empName": "Pauline Kramer",
    "empDept": "HR"
  }
]
-----
[
  {
    "empID": 4,
    "empName": "Jose Hernandez",
    "empDept": "Engineering"
  }
]
-----
[
  {
    "empID": 6,
    "empName": "Pamela Giles",
```

```
"empDept": "Sales"
}]
```

Example: Aggregating Local Results

To do a final union from all AMPs, aggregate the local results from each AMP. To do so, nest a second call to DATASET_PUBLISH. The inner call to DATASET_PUBLISH performs the local aggregation on each AMP, and the outer call does a final aggregation of the local aggregations and produce a single result row that represents all input values.

Use the PARTITION BY clause with a constant value (for example, 1) to perform the final aggregation on a single AMP. By specifying a constant value, like the number 1, the locally aggregated rows from each AMP are in the same partition and redistributed to the same AMP for the final aggregation. The outer query partitions by this constant, shown as "1 as p" in the example. A single output row is returned. You must specify the UNIQUE_NAMES custom clause so that there are no conflicts when combining schemas.

Additionally, reference the aggregated result using dot notation where the recursive descent operator references the inner DATASET_PUBLISH query result, followed by an array reference composed of the * wildcard. This retrieves one array composed of one record per input row. The final query looks similar to:

```
select data.getSchema(), data..record[*] FROM DATASET_PUBLISH
(
  ON (SELECT data as record, 1 as p FROM DATASET_PUBLISH
    (
      ON (SELECT * FROM employeeTable)
      USING UNIQUE_NAMES('Y')
    )as L
  ) partition by p
)G;
```

data.getSchema()	data..record[*]
<pre>{ "type": "array", "items": { "type": "record", "name": "rec_0", "fields": [{ "name": "type": { "type": "name": "record", "record", "rec_0_1448384735",</pre>	<pre>[{ "empID": 5, "empName": "Kyle Newman", "empDept": "Engineering" }, { "empID": 3, "empName": "Steven Mazzo", "empDept": "Engineering" }, { "empID": 1, "empName": "George Smith", "empDept": "Accounting"</pre>

<pre> "fields": [{ "name": "empID", "type": "int" }, { "name": "empName", "type": "string" }, { "name": "empDept", "type": "string" }] } </pre>	<pre> }, { "empID": 2, "empName": "Pauline Kramer", "empDept": "HR" }, { "empID": 6, "empName": "Pamela Giles", "empDept": "Sales" }, { "empID": 4, "empName": "Jose Hernandez", "empDept": "Engineering" }] </pre>
---	---

You can aggregate all values with a single call to DATASET_PUBLISH by partitioning by a constant value. That re-distributes all rows to a single AMP to perform the aggregation, which does not take advantage of the parallel processing capability in the database. By using two calls to DATASET_PUBLISH, the AMPs perform local aggregations in parallel and only the final aggregation is performed on a single AMP.

Avro Object Container Files

About Importing and Exporting

The Avro specification provides a simple object container file format to transmit and store multiple binary-encoded Avro values, along with a common schema.

Although the database does not provide direct support for the files, the following sections describe a general framework you can implement to import or export Avro data to and from these files using the feature's functionality.

Importing From an Avro Object Container File

The Avro specification provides an object container file format to transmit and store multiple binary-encoded Avro values with a common schema.

Because these files contain one Avro schema and one or more binary-encoded Avro values described by that schema, the data in an object container file maps to a DATASET STORAGE FORMAT AVRO column with a column-based schema.

The database provides direct support for the files using the AvroContainerSplit table operator. The following section describes a general framework to import Avro data from the files.

1. Retrieve the schema from the object container file.
2. Create a schema using the new CREATE *<storage-format-name>* SCHEMA DDL statement using the schema retrieved in Step 1. Note that this schema may be specified in LATIN or UNICODE characters or as UTF-8 in its byte representation.
3. Create a table that conforms to a desired structure, and includes a DATASET STORAGE FORMAT AVRO column with a column-level schema defined using the schema created in Step 2.
4. Run the AvroContainerSplit table operator to load the Avro DATASET values into the table created in Step 3.

These steps allow any application to import data from an object container file to a database table.

Note:

If the DATASET table column is defined without a column-based schema, the schema is stored with each Avro instance in the table.

Example: Loading Avro Object Container Files as BLOBs Using BTEQ

The following example shows how to load Avro object container files as BLOBs by using BTEQ. Then, you can extract the Avro values into a table with a column of type DATASET STORAGE FORMAT AVRO by using the AvroContainerSplit table operator.

In this example, three object container files are loaded.

1. Create an import file with two fields for BTEQ called avro_containers.txt. The first field is the container ID, and the second field is the name of the file containing the Object Container file.

The container ID is loaded into an INTEGER column, and the file is loaded as "BLOB as deferred by name" into a BLOB. The import file contents are in vartext format:

```
1|avro_1.db
2|avro_2.db
3|avro_3.db
```

2. Create a table with a BLOB column to import the BLOB data.

```
CREATE TABLE avro_containers(container_id INTEGER, container BLOB);
```

3. Load the table with the object container files from BTEQ.

```
.import vartext file avro_containers.txt

.repeat *
USING (c1 VARCHAR(20), c2 BLOB AS DEFERRED BY NAME) INSERT
INTO avro_containers(:c1, :c2);
```

4. Create a table to hold the Avro values.

```
CREATE TABLE avro_table(container_id INTEGER, avro_obj_id INTEGER, avro
DATASET STORAGE FORMAT AVRO);
```

5. Extract the Avro values from each container by using the AvroContainerSplit table operator.

```
INSERT INTO avro_table
SELECT T.out_container_id, T.avro_object_id, T.avro_value
FROM AvroContainerSplit
  (ON (SELECT container_id, container FROM avro_containers)) T;
```

The Avro values from all three container files are loaded into the table, avro_table, organized by container ID and Avro Object ID within each container ID.

6. Select the first two Avro objects, in JSON format, from the first container.

```
SELECT container_id, avro_obj_id, avro.tojson() FROM avro_table WHERE
container_id = 1 AND
    avro_obj_id < 2
ORDER BY 1,2;
```

Exporting to an Avro Object Container File

Use an application to create an object container file based on data stored in the database.

Use a JDBC application with a Java-based Avro library, or an ODBC application using the C-based Avro library, to construct the object container file. Both applications are available from <https://avro.apache.org/>.

The database files contain one Avro schema and one or more binary-encoded Avro values described by that schema.

The following scenarios use an application to create an object container file based on data stored in the database.

Scenario 1: Data Stored as Avro with Column-Level Schema

Data stored as Avro with a column-level schema maps to an object container file. Follow the steps required to gather the necessary data and construct the object container file:

1. Retrieve the schema used for the column by either selecting it out of the dictionary based on its name, or by selecting it from any instance of the column using the `getSchema` method of the DATASET data type. If the length of the schema is needed as well, the `getSchemaSize` method could be used.
2. Retrieve all of the binary-encoded Avro values from the column using the `getRawData` or `getRawDataLob` methods of the DATASET data type. If the length of these binary-encoded Avro values is needed as well, the `getRawDataSize` method could be used.

Using the schema and binary-encoded Avro values obtained, you can construct an object container file using the Avro libraries.

Scenario 2: Data Stored as Avro Without Column-Level Schema

Use data stored as Avro without a column-level schema to construct an object container file. You have three possibilities for constructing an object container file in this scenario.

The Schemas are the Same, Without a Column-Level Schema

Simply retrieve a schema from an instance, and use this to construct the object container file, following the guidelines of Scenario 1.

The Schemas are Different, But Compatible

Follow the required steps to gather the necessary data and construct the object container file:

1. Determine the desired schema used to describe all binary-encoded Avro values that compose the object container file.
2. Use the `AvroProject` method of the `DATASET` type to construct Avro instances with the desired schema.
3. Retrieve only the binary-encoded Avro values from the instances created in step 2 using the `getRawData` or `getRawDataLob` methods of the `DATASET` data type. If the length of these binary-encoded Avro values is also needed, use the `getRawDataSize` method.

Using the determined upon schema and the projected binary-encoded Avro values, you can construct an object container file using the Avro libraries.

The Schemas are Different, But Not Compatible

Each schema must be isolated, along with any binary-encoded Avro values described by that schema. For each isolated pairing, follow the steps in Scenario 1 to produce an object container file.

Scenario 3: Publishing Data to the Avro Format

Data not stored as Avro, whether stored in a table, a constant value, the result of some other operation, may be published to the Avro format and used to construct an object container file. Follow these steps to gather the data and construct the object container file:

1. Formulate a query to publish data in the desired Avro format using the `DATASET_PUBLISH` table operator.
2. Obtain the schema for the object container file by executing the `getSchema` method of the `DATASET` type on the *data* output column of `DATASET_PUBLISH`. This provides an Avro instance that contains the schema and binary-encoded Avro value created.

This operation only needs to be executed once because the schema produced by `DATASET_PUBLISH` applies to all output instances (and therefore applies to an object container file).

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"> If <i>delimiter</i> is semicolon: x; x; x If <i>delimiter</i> is { , OR }, you can do either of the following: <ul style="list-style-type: none"> x, x, x x OR x OR x 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

External Representations for the DATASET Type

Data Type Encoding

Some of the client and server interfacing parcels such as DataInfo (parcel flavor 71), DataInfoX (parcel flavor 146), PrepInfo (flavor 86), PrepInfoX (parcel flavor 125) and StatementInfo parcel (flavor 169) return a Data Type Code. DataInfo, DataInfoX and StatementInfo can also be used in the client-to-server direction, in which case the application specifies the data type code. For the DATASET data type, the Data Type Code corresponds to the data type of the DATASET data type's transform type. For example, if the transform for a DATASET type in the Avro storage format is a BLOB, the Data Type Code is a BLOB (400 or 401).

For the Server Data Type Code field of the StatementInfo parcel (parcel flavor 169), the following encoding for the DATASET data type is used. The encoding numbers defined follow the pattern for existing data types (such as Nullable number is non-Nullable value + 1, stored procedure IN parameter type number is 500 + non-nullable number, etc.). Note that these type codes are never returned to the client as a Data Type Code, only as a Server Data Type Code.

Data Type	NULL Property		Stored Procedure Parameter Type		
	Non-nullable	Nullable	IN	INOUT	OUT
DATASET STORAGE FORMAT Avro	512	513	1012	1013	1014

These codes are sent from server to client, and are accepted by server from client in the parcels described in the following sections. The only restriction is the type may not be used in the USING clause. VARBYTE/BLOB can be used for Avro and when necessary, this data will be implicitly casted to the DATASET type.

SQL Capabilities Parcel

The SQL Capabilities parcel includes a flag called SQLCap_DATASET which indicates whether the DATASET data type is supported in the database.

```
typedef
struct PclCfgSQLCapFeatType {
    PclCfgFeatureType      SQLCap_Feature;
    PclCfgFeatureLenType   SQLCap_Length;
    byte /* 0 */           SQLCap_UPSERT;
    byte /* 1 */           SQLCap_ArraySupport;
    .
    .
    .
    byte /* 20 */          padbyte_boolean;
```

```

        byte /* 21 */          SQLCap_JSON;
        byte /* 24 */ SQLCap_DATASET;
    } PclCfgSQLCapFeatType;

```

The SQLCap_DATASET flag has the following values:

- 0 indicates that the DATASET data type is not supported.
- 1 indicates that the DATASET data type is supported.

Database Limits (ConfigResponse) Parcel

The Database Limits Parcel did not change. The maximum bytes allowed in a DATASET data type instance are governed by the value of:

```

+12 (64-bit int)          -Max. bytes in LOB

```

StatementInfo Parcel

The StatementInfo Parcel did not change.

The DATASET data type is considered a native data type. The following fields of the StatementInfo Parcel contain information relevant to a particular instance of the DATASET data type:

- Data Type Code = VARBYTE or BLOB
- UDT indicator = 0 (DATASET data type is treated as a system built-in type)
- Fully qualified type name length = 0
- Fully qualified type name = ""
- Max Length in Bytes = The maximum possible length in *bytes* for this particular DATASET instance
- Server Data Type Code: Data type code is DATASET STORAGE FORMAT Avro.

Example: Metadata Parcel Sequence

This example shows a statement and the associated Metadata Parcel Sequence.

Consider the following table, data, and query.

```

CREATE SET TABLE myDatasetTable ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
    DEFAULT MERGEBLOCKRATIO
(
    id INTEGER,

```



```
avroFile DATASET(543000) STORAGE FORMAT Avro)
PRIMARY INDEX ( id );
```

When executing the SELECT statement, the StatementInfo parcel looks like the following:

Database Name	test_db
Table/View Name	myDatasetTable
Column Name	avroFile
Column Index	3
As Name	
Title	avroFile
Format	
Default Value	
Is Identity Column	N
Is Definitely Writable	Y
Is Nullable	Y
Is Searchable	Y
Is Writable	Y
Data Type Code	400 or 688 (depending on transform)
UDT Indicator	0
UDT Name	
UDT Misc	
Maximum transform size	543000
Digits	0
Interval Digits	0
Fractional Digits	0
Max Number of Characters	0
Is CaseSensitive	N
Is Signed	U
Is Key Column	N
Is Unique	N
Is Expression	N

Is Sortable	N
Parameter Type	U
Struct Depth	0
Is Temporal Column	0
UDT Attribute Name	
Server Data Type Code	517 (DATASET STORAGE FORMAT Avro, nullable)
Array Number of Dims	0

Additional Information

Teradata Links

Link	Description
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none">• Community support• Software updates• Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community