TERADATA®

What would you do if you knew?™

Teradata Database

Teradata DATASET Data Type

Release 16.00
B035-1198-160K
December 2016

TERADATA LABS®

Preface

Purpose

Teradata DATASET Data Type describes support for DATATSET data. DATASET is a complex data type 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.

Audience

Database administrators, application programmers, and other technical personnel responsible for designing, maintaining, and using the Teradata Database.

Supported Software Releases and Operating Systems

This book supports Teradata® Database 16.00.

Teradata Database 16.00 is supported on:

- SUSE Linux Enterprise Server (SLES) 11 SP1
- SUSE Linux Enterprise Server (SLES) 11 SP3

Teradata Database client applications support other operating systems.

Changes to This Book

Date	Release	Description
December 2016	16.00	Initial publication.

Additional Information

URL	Description
http://www.info.teradata.com	Use the Teradata Information Products Publishing Library site to:

URL	Description
	• View or download a manual: 1. Under **Online Publications**, select **General Search**. 2. Enter your search criteria and click **Search**. • Download a documentation CD-ROM: 1. Under **Online Publications**, select **General Search**. 2. In the **Title or Keyword** field, enter *CD-ROM*, and click **Search**.
www.teradata.com	The Teradata home page provides links to numerous sources of information about Teradata. Links include: • Executive reports, white papers, case studies of customer experiences with Teradata, and thought leadership • Technical information, solutions, and expert advice • Press releases, mentions and media resources
www.teradata.com/TEN/	Teradata Customer Education delivers training that builds skills and capabilities for our customers, enabling them to maximize their Teradata investment.
https://tays.teradata.com	Use Teradata @ Your Service to access Orange Books, technical alerts, and knowledge repositories, view and join forums, and download software patches.
Teradata Community - Developer Zone	Developer Zone provides new ideas and developer information.
Teradata Downloads	Provides publicly available downloads from Teradata.

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please email teradata-books@lists.teradata.com.

Product Safety Information

This document may contain information addressing product safety practices related to data or property damage, identified by the word *Notice*. A notice indicates a situation which, if not avoided, could result in damage to property, such as equipment or data, but not related to personal injury.

Example

Notice:
Improper use of the Reconfiguration utility can result in data loss.

Teradata Database Optional Features

This book may include descriptions of the following optional Teradata Database features and products:

- In-Memory Optimization
- Teradata Columnar

- Teradata Row-Level Security
- Teradata Secure Zones
- Teradata Temporal
- Teradata Virtual Storage (VS)

You may not use these features without the appropriate licenses. The fact that these features may be included in product media or downloads, or described in documentation that you receive, does not authorize you to use them without the appropriate licenses.

Contact your Teradata sales representative to purchase and enable optional features.

CHAPTER 1

The DATASET Data Type

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 Teradata Database.

Function	Description
Storage and processing	• Store variable data formats. The Avro format is currently the only format 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	• 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])
- Index slices ([0:5])

Client Support for the DATASET Data Type

Client Product	DATASET Support Provided
CLI	Full native DBS support.

Client Product	DATASET Support Provided
ODBC	• 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	• 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 Teradata 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	• 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 Teradata Database. • Schema Collections (Data Dictionary) also support the DATASET data type.
Teradata Parallel Transporter (TPT)	DATASET columns are treated like 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. The Avro format is an example 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 Teradata Database stores and operates on data in its native format using dot notation.

Overview of the DATASET Data Type

DATASET is a complex data type provided by the Teradata Database, 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 Teradata Database in an optimized format depending on the size of the data.

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.

Note:
Teradata Database only supports using CSV in some functions and table operators provided, not as a storage format of the DATASET type.

DATASET adds the following non-reserved keywords:

- DOT
- NOTATION
- LIST
- AVRO
- CREATEDATASET
- 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 to provide variability in maximum and in-line lengths. The values represent the size in bytes of the schema and data for each instance of the DATASET data type.

Additional space is reserved for context information for a particular instance, but the specified values only apply to the schema and data.

Minimum or Maximum Length Type	Length Size
Maximum LOB length	2 GB
Maximum row size	64 K
Default maximum length (if users do not provide length)	2 GB
Default inline length (if users do not provide length)	10 K
Minimum length	100 bytes
Maximum size of an Avro schema that is not column-based	16 MB
Maximum size of a binary-encoded Avro 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. Teradata Database 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

Currently, Teradata Database provides the Avro storage format for use with the DATASET data type. The Avro storage format requires that specific requirements must be met, based on the Apache Avro specification. Each instance must contain a schema specified according to the Avro specification. The schema is interpreted on a per-instance basis, or at the column level.

Storage Format Terminology

Term	Description
Schema	A JSON document used to describe the format of the binary-encoded Avro value. It can be specified in JSON text, but the JSON text may be specified in UTF-8 encoded characters using a VARBYTE or BLOB data type.
Binary-encoded Avro Value	The actual Avro data, encoded according to the scheme described by the Avro schema.

Term	Description
JSON-encoded Avro Value	JSON-text representation of the data, as described by the Avro schema.
Transform format OR Cast format	A null-terminated, UTF-8 encoded schema followed immediately by a binary-encoded Avro value.

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.

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:

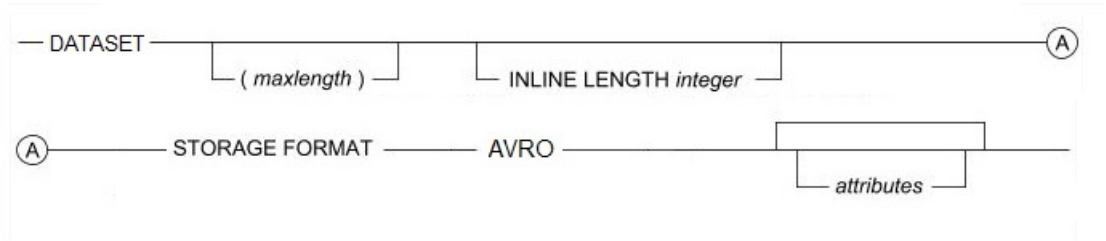
- Privilege: WITH DATASET SCHEMA
- Abbreviation in System Views: W1
- Automatically Granted
  - Creators: Yes
  - Created User or Database: No
- Explicitly Granted
  - Creators: Yes
  - Created User or Database: Yes
  - Privilege Category: Dataset Schema

For more information, see *SQL Data Control Language*.

DATASET Data Type Syntax

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.



Syntax Elements

maxlength

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

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.

AVRO

The built-in storage format.

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 *SQL Data Types and Literals*.

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 *SQL Data Definition Language*.

Character Set Handling

The Avro specification requires that character strings be stored as UTF-8 encoded character data. Although the Teradata 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.

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 imports and exports DATASET data from a client system to the Teradata Database, and from Teradata Database to a client system.

Complex data types use multiple transforms, after specifying which system-provided transform is used by default for a certain CDT. DATASET provides transform groups, based on the DATASET type instance storage format.

STORAGE FORMAT Avro transforms must meet the following requirements:

- Transform to/from BLOB via 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 via TD_DATASET_AVRO_VARBYTE

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

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

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

No other casts are supported.

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

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.

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

UDFs

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 TypeId 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 "DATASET_AVRO_DT". The type codes for dtype_en include:

```
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,
        XML_DT = 40,
        ST_GEOMETRY_DT = 41,
        MBR_DT = 42,
        MBB_DT = 43,
        ARRAY_DT = 44,
        JSON_DT = 45,
      DATASET_AVRO_DT = 46,
        FNC_DATATYPESETSIZE=47
} dtype_en;
```

The complex types map to the following base type, DATASET_AVRO_DT → BLOB_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 size of the data
TD_DATASET_AVRO_BLOB	BLOB_REFERENCE_DT	<name of the column>		The size of the data

The data type provides no "transforms off" functionality. The UDT_BaseInfo_t structure's udt_indicator member has a value to identify the storage formats of DATASET:

10==DATASET STORAGE FORMAT AVRO

XSPs

The CREATE/REPLACE PROCEDURE statement was extended to create an external stored procedure (XSP) 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 XSP 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.
FNC_GetDatasetResultLob	Writes DATASET data to a LOB associated with any DATASET instance.
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.
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

FNC Library Routine	Description
	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.

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 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 details about the DATASET type interface functions, see *SQL External Routine Programming*.

Restrictions for the DATASET Type

There are length restriction 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

DATASET currently provides one built-in storage format, Avro.

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/ALTER TABLE statement as an attribute of a DATASET data type column AND the schema was previously registered via the CREATE <storage-format-name> 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.

Data is always validated against the schema provided for a particular instance.

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/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 pre-defined, comparable type. Because the type may not be compared, it is not used for comparison in a SET table.

CHAPTER 2

Operations on the DATASET Data Type

Creating and Altering Tables to Store DATASET Data

Purpose

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

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
);
```

Using Algorithmic Compression on DATASET Columns

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

The DATASET data type provides a set of compression and decompression algorithms, SNAPPY_COMPRESS and SNAPPY_DECOMPRESS, for DATASET STORAGE FORMAT AVRO columns. For the algorithms, use standard algorithmic compression syntax with the routines:

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

For more information, see *SQL Data Types and Literals*.

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])
- index slices ([0:5])

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:

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

avroFiles09.txt
avro09.data|1

avro09.data
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F7264
222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D65223A224974656
D5F4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C
2274797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F72222C227
4797065223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C227479
7065223A22737472696E67227D2C7B226E616D65223A225175616E746974795F507572636861736
564222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C22
74797065223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C227
4797065223A22646F75626C65227D5D7D7D5D7D006E0E62696379636C650672656408626F797302
00000000000059400000000000005940

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

Example: Using Dot Notation with the DATASET Data Type

Using the table Myavrotable09:

```
/*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;

>    {
       "Item_ID" : 55,
        "Item_Name" : "bicycle",
       "Item_Color" : "red",
       "Item_Style" : "boys",
       "Quantity_Purchased" : 1,
       "Item_Price" : 100.00,
       "Total_Price" : 100.00
    }

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

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

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

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");
*** 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, which was created and loaded as a source table:

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

Example: Successful INSERT-SELECT Statements

The following are examples of successful INSERT-SELECT statements:

```
CREATE TABLE myAVROTable02(
    id INT,
    avroFile DATASET STORAGE FORMAT Avro);
INSERT INTO myAVROTable02 SELECT * FROM myAVROTable01;
```

Example: Unsuccessful INSERT-SELECT Statements

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

```
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
```

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
- ExtractValue
- getRawData
- getRawDataLob
- getRawDataSize
- getSchema
- getSchemaSize
- toJSON
- Validate

Functions and Table Operators

- AVRO_CHECK
- AvroContainerSplit
- CreateDATASET
- Dataset_Keys
- DATASET_TABLE
- SchemaEqual
- SchemaMatch

CHAPTER 3
DATASET Methods

AvroProject

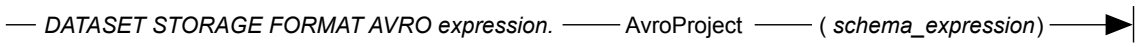
Purpose

AvroProject allows for reading specified portions of an Avro instance, without having to read the entire instance.

For example, take an Avro instance that has a 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.

Syntax



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.

Rules and Restrictions

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

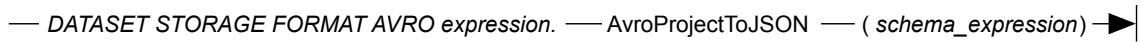
```
CREATE TABLE avroTable(id INTEGER, avroCol DATASET STORAGE FORMAT AVRO);
/*insert some data composed of a record with five fields as mentioned above*/
/*
{"id":1,"First":"Leo","Last":"Tolstoy","Phone":"(800)-123-4657","Age":187}
*/
INSERT INTO avroTable(1,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A224
669727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22
74797065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A2
2737472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D00
02064C656F0E546F6C73746F791C28383030292D3132332D34363537F602'xb);
/*
{"id":2,"First":"Mark","Last":"Twain","Phone":"(800)-123-4657","Age":180}
*/
INSERT INTO avroTable(2,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A224
669727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22
74797065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A2
2737472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D00
04084D61726B0A547761696E1C28383030292D3132332D34363537E802'xb);
/*
{"id":3,"First":"William","Last":"Shakespeare","Phone":"(800)-123-4657","Age":
451}
*/
INSERT INTO avroTable(3,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A224
669727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22
74797065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A2
2737472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D00
060E57696C6C69616D165368616B657370656172651C28383030292D3132332D343635378607'xb
);
/*
{"id":4,"First":"Charles","Last":"Dickens","Phone":"(800)-123-4657","Age":203}
*/
INSERT INTO avroTable(4,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226964222C2274797065223A22696E74227D2C7B226E616D65223A224
669727374222C2274797065223A22737472696E67227D2C7B226E616D65223A224C617374222C22
74797065223A22737472696E67227D2C7B226E616D65223A2250686F6E65222C2274797065223A2
2737472696E67227D2C7B226E616D65223A22416765222C2274797065223A22696E74227D5D7D00
080E436861726C65730E4469636B656E731C28383030292D3132332D343635379603'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

Purpose

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

Syntax



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.

Return Value

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

Rules and Restrictions

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                                           {"First":"Leo","Age":187}
```

2	{"First":"Mark","Age":180}
3	{"First":"William","Age":451}
4	{"First":"Charles","Age":203}

ExtractValue

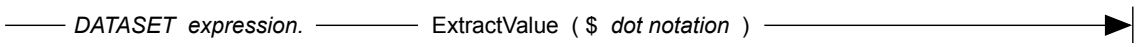
Purpose

Some queries on a DATASET instance are not possible using dot notation. These queries might contain a dot notation expression too large to be expressed in Teradata-SQL, or use characters or words not permitted in the syntax.

Additionally, some queries have results that are too large to be returned by a dot notation expression.

The ExtractValue method is provided for these types of queries.

Syntax



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])
- Index slices ([0:5])

Rules and Restrictions

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.

Example: Returning Values If the Query Cannot Be Properly Expressed In Dot Notation

Populate the myAVROTable06 table:

```
avro08b.data
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F7264
222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D65223A224974656
D5F4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C
2274797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F72222C227
4797065223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C227479
7065223A22737472696E67227D2C7B226E616D65223A225175616E746974795F507572636861736
564222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C22
74797065223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C227
4797065223A22646F75626C65227D5D7D7D5D7D006E0E62696379636C650672656406 26F797302
0000000000005940000000000000005940
CREATE TABLE myAVROTable06(
    id INTEGER,
    avroFile DATASET STORAGE FORMAT Avro);
.import vartext file avro08.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.

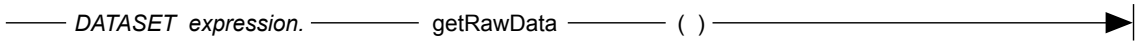
```
SELECT avroFile.ExtractValue('$.Sale.Item_ID') FROM myAVROTable06;
    > 55

    SELECT avroFile.ExtractValue('$.Sale[Item_ID,Item_Name]') FROM
myAVROTable06;
    > [55,"bicycle"]
```

getRawData

Purpose

getRawData retrieves the raw data of any DATASET data type instance, returning as a non-LOB type.

Syntax



Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Return Value

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

Rules and Restrictions

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

Example: Retrieving the Raw Data of Any DATASET Data Type Instance and Returning it as a Non-LOB Type

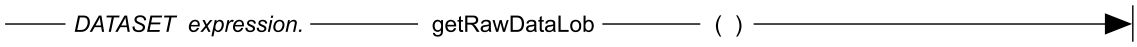
```
SELECT cast(avroFile.getRawData() as varbyte(5000))
    FROM myAVROTable06;
> 6E0E62696379636C650672656408626F797302
  00000000000059400000000000005940
```

getRawDataLob

Purpose

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

Syntax



Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Return Value

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

Rules and Restrictions

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.

Example: Retrieving the Raw Data of Any DATASET Data Type Instance

```
getRawDataLob
SELECT avroFile.getRawDataLob() FROM myAVROTable06;
> 6E0E62696379636C650672656408626F797302
  0000000000005940000000000005940
```

getRawDataSize

Purpose

getRawDataSize retrieves the size of the raw data for any instance of the DATASET data type.

Syntax

Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Return Value

The result is in terms of bytes, not characters.

Example: Retrieving the Size of the Raw Data for Any Instance of the DATASET Data Type

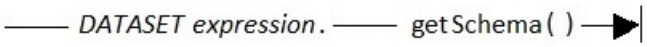
```
SELECT avroFile.getRawDataSize() FROM myAVROTable06;
> 35
```

getSchema

Purpose

getSchema retrieves the schema of any instance of the DATASET data type.

Syntax



Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Return Value

getSchema provides a JSON text representation (max length, character set UNICODE) of the schema for any instance of the DATASET data type which uses any storage format.

Example: Retrieving the Schema of Any 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"}
                ]
                }
            }
       ]
     }
```

getSchemaSize

Purpose

getSchemaSize retrieves the size of the schema for any instance of a DATASET data type.

Syntax

Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Functional Description

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

Return Value

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.

Example: Retrieving the Size of the Schema for Any Instance of a DATASET Data Type

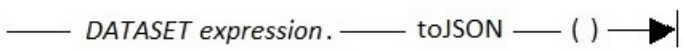
```
SELECT avroFile.getSchemaSize() FROM myAVROTable06;
> 375
```

toJSON

Purpose

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

Syntax



Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Return Value

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

Example: Converting Any Instance of the DATASET Data Type into a JSON Text Instance

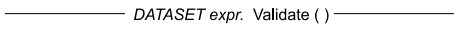
```
SELECT avroFile.toJSON() FROM myAVROTable06;
>{
        "Sale" : {
            "Item_ID" : 55,
            "Item_Name" : "bicycle",
            "Item_Color" : "red",
            "Item_Style" : "boys",
            "Quantity_Purchased" : 1,
            "Item_Price" : 100.00,
            "Total_Price"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

Validate

Purpose

Invoke the Validation method on a DATASET type instance to report back whether data is valid.

Syntax



Syntax Elements

DATASET expression

Any expression that evaluates to a DATASET data type.

Return Value

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

Example: Validating a DATASET Type Instance

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
7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C64732
23A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A227265636F7264
222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E616D65223A224974656
D5F4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F4E616D65222C
2274797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6C6F72222C227
4797065223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C65222C227479
7065223A22737472696E67227D2C7B226E616D65223A225175616E746974795F507572636861736
564222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F5072696365222C22
74797065223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F5072696365222C227
4797065223A22646F75626C65227D5D7D7D5D7D006E0E62696379636C650672656408626F797302
0000000000005940000000000000
/*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

CHAPTER 4

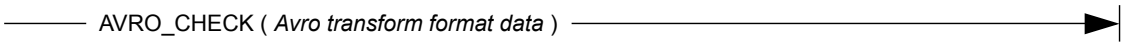
DATASET Functions and Operators

AVRO_CHECK

Purpose

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.

Syntax



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.

Rules and Restrictions

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
('002274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C647
3223A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002'xb);
> Invalid Avro Schema: Unexpected end-of-input
```

Example: Passing Invalid Data

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

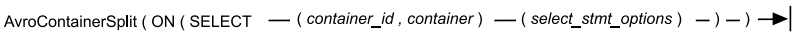
```
SELECT
AVRO_CHECK('002274797065223A227265636F7264222C226E616D65223A227265635F30222C226
669656C6473223A5B7B226E616D65223A2261222C2274797065223A22696E74227D5D7D0002'xb)
;
> Invalid Avro Schema: Unexpected end-of-input
```

AvroContainerSplit

Purpose

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

Syntax



Syntax Elements

container_id

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

container

A BLOB in the Avro Object Container File format.

select_stmt_options

The allowable or necessary options in Teradata Database for a SQL select statement.

Rules and Restrictions

The input data to the AvroContainerSplit table operator, as determined by the result of the SELECT statement in the ON clause, must be one or more rows of data, each with two columns defined as follows:

- *container_id* includes the following types:
  - BYTEINT
  - SMALLINT
  - INTEGER
  - BIGINT
  - DOUBLE PRECISION
  - DECIMAL
  - NUMBER
  - CHAR
  - VARCHAR
- *container*

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.

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()
```

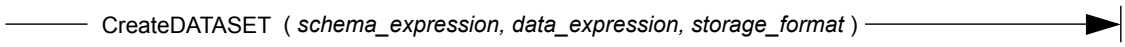
```
FROM avro_table
ORDER BY container_id, avro_id;
```

CreateDATASET

Purpose

The CreateDATASET function allows for creating DATASET data type instances composed of self-describing data when the schema and data are not included in the data payload.

Syntax



Syntax Elements

schema_expression

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

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

data_expression

Any expression that evaluates to Teradata BYTE/VARBYTE/BLOB (for STORAGE FORMAT AVRO) data.

storage_format

The name of the storage format. Avro is the only storage format currently supported.

Rules and Restrictions

CreateDATASET is used to create DATASET data type instances. The instances are composed of self-describing data where the schema and data arrive separately. If the data is already with its schema, you do not need CreateDATASET to create an instance from the data.

CreateDATASET has three required parameters:

- Schema: A schema conforming to the specifications. 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: A BYTE/VARBYTE/BLOB containing the binary-encoded self-describing data value. Data can be specified in any way Teradata Database 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.
- Storage format: Specify the name of the storage format. Any other value, including a null or omitted value, are invalid.

Examples

Example: Creating DATASET Data Type Instances

In the 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
7B2274797065223A226172726179222C226974656D73223A5B7B2274797065223A227265636F726
4222C226E616D65223A227265635F30222C226669656C6473223A5B7B226E616D65223A2253616C
65222C2274797065223A7B2274797065223A227265636F7264222C226E616D65223A227265635F3
1222C226669656C6473223A5B7B226E616D65223A224974656D5F4944222C2274797065223A2269
6E74227D2C7B226E616D65223A224974656D5F4E616D65222C2274797065223A22737472696E672
27D2C7B226E616D65223A224974656D5F436F6C6F72222C2274797065223A22737472696E67227D
2C7B226E616D65223A224974656D5F5374796C65222C2274797065223A22737472696E67227D2C7
B226E616D65223A225175616E746974795F5075726368617365644222C2274797065223A22696E74
227D2C7B226E616D65223A224974656D5F5072696365222C2274797065223A22646F75626C65227
D2C7B226E616D65223A22546F74616C5F5072696365222C2274797065223A22646F75626C65227D
5D7D7D5D7D2C7B2274797065223A227265636F7264222C226E616D65223A227265635F32222C226
669656C6473223A5B7B226E616D65223A2253616C65222C2274797065223A7B2274797065223A22
7265636F7264222C226E616D65223A227265635F33222C226669656C6473223A5B7B226E616D652
23A224974656D5F4944222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F
4E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A224974656D5F436F6
C6F72222C2274797065223A22737472696E67227D2C7B226E616D65223A224974656D5F5374796C
65222C2274797065223A226E756C6C227D2C7B226E616D65223A225175616E746974795F5075726
36861736564222C2274797065223A22696E74227D2C7B226E616D65223A224974656D5F50726963
65222C2274797065223A22646F75626C65227D2C7B226E616D65223A22546F74616C5F507269636
5222C2274797065223A22646F75626C65227D5D7D7D5D7D2C7B2274797065223A227265636F7264
222C226E616D65223A227265635F34222C226669656C6473223A5B7B226E616D65223A2253616C6
5222C2274797065223A7B2274797065223A227265636F7264222C226E616D65223A227265635F35
222C226669656C6473223A5B7B226E616D65223A224974656D5F4944222C2274797065223A22696
E74227D2C7B226E616D65223A224974656D5F4E616D65222C2274797065223A22737472696E6722
7D2C7B226E616D65223A224974656D5F436F6C6F72222C2274797065223A226E756C6C227D2C7B2
26E616D65223A224974656D5F5374796C65222C2274797065223A226E756C6C227D2C7B226E616D
65223A225175616E746974795F5075726368617365644222C2274797065223A22696E74227D2C7B2
26E616D65223A224974656D5F5072696365222C2274797065223A22646F75626C65227D2C7B226E
616D65223A22546F74616C5F5072696365222C2274797065223A22646F75626C65227D5D7D7D5D7
D5D7D|
06006E0E6269637636C650672656408626F797302000000000005940000000000005940026E1
0746F7920626F61740870696E6B023333333333332E403333333333332E4004D20108736F617002
```

```
AE47E17A14AEEF3FAE47E17A14AEEF3F00|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
        }}}
    ]
```

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
6E0E62696379636C650672656408626F797302000000000005940000000000005940|1
6E0E62696379636C650672656408626F797302000000000005940000000000005940|2
6E0E62696379636C650672656408626F797302000000000005940000000000005940|3
6E0E62696379636C650672656408626F797302000000000005940000000000005940|4
6E0E62696379636C650672656408626F797302000000000005940000000000005940|5
6E0E62696379636C650672656408626F797302000000000005940000000000005940|6
6E0E62696379636C650672656408626F797302000000000005940000000000005940|7
6E0E62696379636C650672656408626F797302000000000005940000000000005940|8
6E0E62696379636C650672656408626F797302000000000005940000000000005940|9
6E0E62696379636C650672656408626F797302000000000005940000000000005940|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;
```

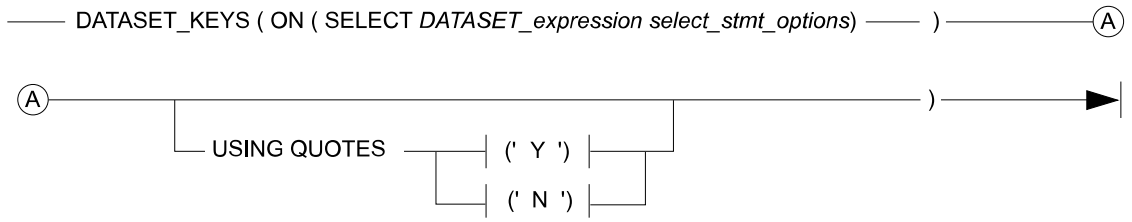
id	avroCol.AvroProject(…)
1	{ "Sale" : { "Item_ID" : 55, "Item_Name" : "bicycle", "Item_Color" : "red", "Item_Style" : "boys", "Quantity_Purchased" : 1, "Item_Price" : 100.00, "Total_Price" : 100.00 } }

DATASET_KEYS

Purpose

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.

Syntax



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 Teradata Database for a SQL select statement.

USING clause

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).

Rules and Restrictions

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.

Example: Using DATASET_KEYS

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

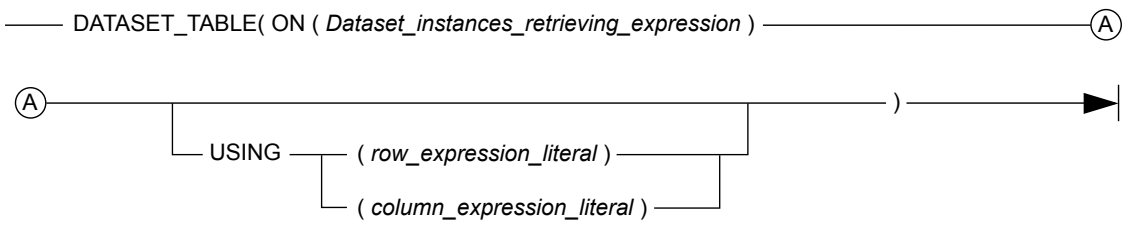
```
Dataset_Keys
/*simple key extraction*/
SEL * FROM DATASET_KEYS
(
    ON (SELECT avroFile FROM myAVROTable09)
) AS avroKeys ORDER BY 1;
> Item_ID
  Item_Name
  Item_Color
  Item_Style
  Quantity_Purchased
  Item_Price
  Total_Price

/*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"
```

DATASET_TABLE

Purpose

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.

Syntax



Syntax Elements

Dataset_instances_retrieving_expression

A query expression or a table or view name resulting in DATASET instances that gets the instances to be shredded.

There must be at least two result columns. One column is an ID that identifies a DATASET instance, and is used to identify a problematic instance, if encountered. The other column is the DATASET instance itself. Other columns can be included, and must 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 result of the function is a table with no rows.

USING clause

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.

column_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 or of the row expression result (depending on the presence or absence of the 'fromRoot' attribute). The *column_expression_literal parameter* defines the columns defined for the output table, and where to get data to populate the columns.

If this parameter is null, an error occurs.

Rules and Restrictions

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

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

The ColExpr parameter requires mapping the columns in the RowExpr to the output table columns for the function. The column data type of the output table can be any non-LOB predefined Teradata type, and must be specified in the ColExpr. Each column in the ColExpr is defined by a JSON object that must conform to one of the following structure two structures: 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 }
```

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.

```
CREATE TABLE my_table (id INTEGER, DatasetCol DATASET STORAGE FORMAT AVRO);
INSERT INTO my_table (1,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616
D65223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73
222C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B22747970652
23A227265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E61
6D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797
065222C2274797065223A22737472696E67227D5D7D7D7D2C7B226E616D65223A226A6F62222C22
74797065223A22737472696E67227D5D7D000E43616D65726F6E3008084C616B6514656C656D656
E746172790E4D616469736F6E0C6D6964646C650C52616E63686F086869676806554349 0E636F6C
6C656765001470726F6772616D6D6572'xb);
INSERT INTO my_table (2,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616
D65223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73
222C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B22747970652
23A227265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E61
6D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797
065222C2274797065223A22737472696E67227D5D7D7D7D5D7D000E4D656C697373612E08084C61
6B6514656C656D656E746172790E4D616469736F6E0C6D6964646C650C52616E63686F086869676
8144D69726120436F7374610E636F6C6C65676500'xb);
INSERT INTO my_table (3,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616
D65223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73
222C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B22747970652
23A227265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E61
```

```
6D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797
065222C2274797065223A22737472696E67227D5D7D7D7D2C7B226E616D65223A226A6F62222C22
74797065223A22737472696E67227D5D7D0008416C65783208084C616B6514656C656D656E74617
2790E4D616469736F6E0C6D6964646C650C52616E63686F08686967680A435355534D0E636F6C6C
6567650006435041'xb);
INSERT INTO my_table (4,
'7B2274797065223A227265636F7264222C226E616D65223A227265635F30222C226669656C6473
223A5B7B226E616D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616
D65223A22616765222C2274797065223A22696E74227D2C7B226E616D65223A227363686F6F6C73
222C2274797065223A7B2274797065223A226172726179222C226974656D73223A7B22747970652
23A227265636F7264222C226E616D65223A227265635F31222C226669656C6473223A5B7B226E61
6D65223A226E616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A2274797
065222C2274797065223A22737472696E67227D5D7D7D7D2C7B226E616D65223A226A6F62222C22
74797065223A22737472696E67227D5D7D000A4461766964320608 4C616B6514656C656D656E746
172790E4D616469736F6E0C6D6964646C650C52616E63686F086869676 80028736D616C6C206275
73696E657373206F776E6572'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
```

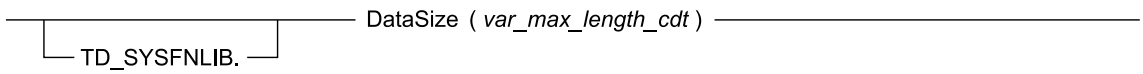
DataSize

Purpose

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

- DATASET
- JSON
- ST_Geometry
- XML

Syntax



Syntax Elements

var_max_length_cdt

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

Return Value

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

Examples: DataSize

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
```

SchemaMatch

Purpose

The SchemaMatch function compares CHAR/VARCHAR/CLOB/JSON types used to represent a schema for compatibility.

Syntax

Syntax Elements

schema_expression

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

Rules and Restrictions

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",
```

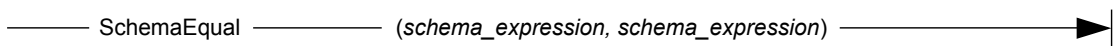
```
    "doc": "this is unique01",
    "fields" : [
        {"name": "a", "type": "long"},
        {"name": "b", "type": "string"}
    ]
 }}');
> 0

/*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
```

SchemaEqual

Purpose

The SchemaEqual function compare CHAR/VARCHAR/CLOB/JSON types representing a self-describing schema for equality.

Syntax



Syntax Elements

schema_expression

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

Rules and Restrictions

Note:
These rules are for the Avro schema, the only storage type currently supported. As more storage formats become available, rules for those formats will be defined.

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
```

CHAPTER 5

CSV Conversion

Overview of 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. This chapter describes the various table operators that allow for conversion.

Examples in the chapter use the following table several times:

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

csv09.txt (note that the record delimiter is shown as '\r\n')
Item_ID,Item_Name,Item_Color,Item_Style,Quantity_Purchased,Item_Price,Total_Pri
ce \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

CSV is not a supported storage format of the DATASET type, but is used with some functions and table operators provided with the DATASET data type.

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

- field_delimiter:

  Specify to define the characters which delimit a field in the CSV data. Use any character or set of characters supported in the chosen character set (Teradata LATIN or UNICODE) for CSV delimiters. If the field delimiter character must be present in the CSV data field, wrap the entire field in double quotes.

- record_delimiter:

  Specify to define the characters which delimit a record in the CSV data. Use any character or set of characters supported in the chosen character set (Teradata LATIN or UNICODE) for CSV delimiters. If the record delimiter character must be present in the CSV data field, wrap the entire field in double quotes.
- field_names:

  Specify to define the names of each field in the CSV instance. If specified, it must be composed of a JSON array of strings containing the same number of values as fields present in the CSV instance. If specified and the data contains the header line, the header line is treated as the first line of data. The database cannot interpret the CSV data to determine what is or is not a header, so if included in the schema, it assumes there is no header line in the data.

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

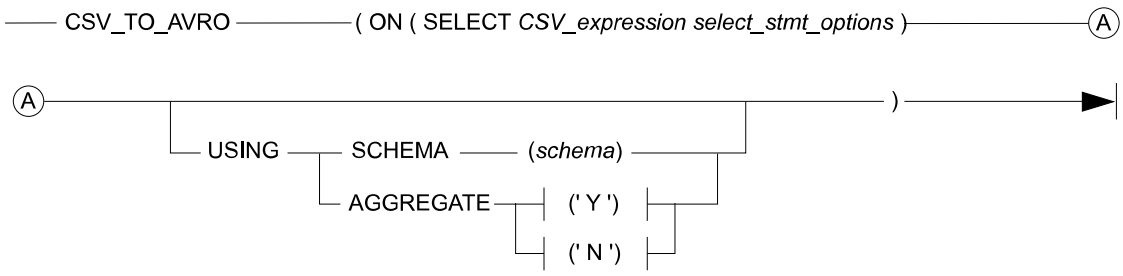
```
Item ID&Item Name&Item Color&Item Style&Quantity Purchased&Item Price&Total
Price#55&bicycle&red&boys&1&100.00&100.00#88&toy
boat&pink&&1&15.10&15.10#105&soap&&&1&0.99&0.99
```

CSV_TO_AVRO

Purpose

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

Syntax



Syntax Elements

CSV_expression

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

select_stmt_options

The allowable or necessary options in Teradata for a SQL select statement.

USING clause

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.

Specifying Input Values

The first input value specified via the ON clause must be character based (for example, CHAR/VARCHAR/CLOB) 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":"\\"}')
) 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 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 instance. All values are treated as strings, except the NULL fields are converted to null Avro 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_Pri
ce#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,"Quanti
ty_Purchased":"1","Item_Price":"0.99","Total_Price":"0.99"}
{"Item_ID":"55","Item_Name":"bicycle","Item_Color":"red","Item_Style":"boys","Q
uantity_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_Pric
e":"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)
    USING SCHEMA('{"record_delimiter":"#", "field_names":
["a1","a2","a3","a4","a5","a6","a7"]}')
) AS csvAvro ORDER BY avroData."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":"1
00.00"}
> {"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":"Quan
tity_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. In the example, *"Item_Color"* allows record or field delimiters to be present in CSV data keys or values.

```
SELECT data.toJSON() FROM CSV_TO_AVRO
(
    ON (SELECT
'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')
    USING SCHEMA('{"record_delimiter":"_"}')
) AS csvAvro ORDER BY data."Item_ID";

> {"Item":"ID","col_1":"Item"}
> {"Item":"Name","col_1":"Item_Color","col_2":"Item"}
> {"Item":"Style","col_1":"Quantity"}
> {"Item":"Purchased","col_1":"Item"}
> {"Item":"Price","col_1":"Total"}
> {"Item":"\nPrice"}
>
{"Item":"55","col_1":"bicycle","col_2":"red","col_3":"boys","col_4":"1","col_5"
:"100.00","col_6":"100.00"}
> "Item":"88","col_1":"toy
boat","col_2":"pink","col_3":null,"col_4":"\n1","col_5":"15.10","col_6":"15.10"
}
>
{"Item":"105","col_1":"soap","col_2":null,"col_3":null,"col_4":"1","col_5":"0.9
9","col_6":"0.99"}
```

Example: Aggregating Avro Output

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

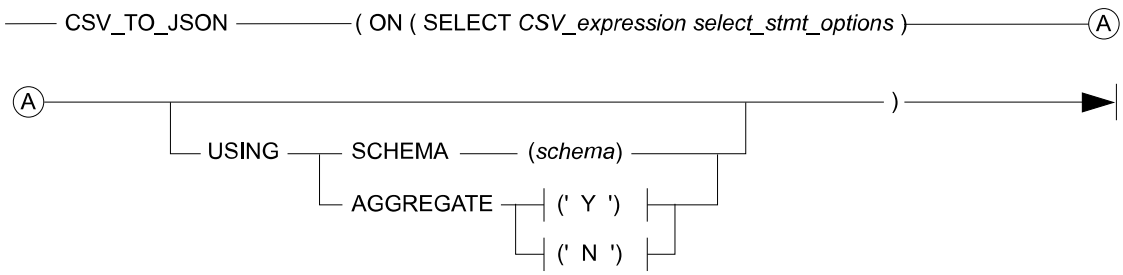
```
SELECT data.toJSON() FROM CSV_TO_AVRO
 (
    ON (SELECT csvFile FROM myCSVTable09 WHERE id=2)
    USING SCHEMA('{"record_delimiter":"#"}')DO_AGGREGATE('Y')
) AS csvAvro;

> [{"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"}]
```

CSV_TO_JSON

Purpose

The CSV_TO_JSON table operator converts CSV data into JSON data.

Syntax



Syntax Elements

CSV_expression

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

select_stmt_options

The currently allowable or necessary options in Teradata for a SQL select statement.

USING clause

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.

Specifying Input Values

The first input value specified via the ON clause must be character based (for example, CHAR/VARCHAR/CLOB) 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 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: Using the SCHEMA Custom Clause to Specify Non-Standard Data

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

```
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. 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_
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')
    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"}]
```

CHAPTER 6

DATASET Publishing

About Publishing

The Teradata 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 data stored within Teradata 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 via DATASET_PUBLISH, its schema is combined with the output schema.
- 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 instance via DATASET_PUBLISH are converted to their text representation and treated as one field in the resulting Avro instance. Note that distinct and structured UDTs are not supported.

The DATASET_PUBLISH table operator uses three 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.

Additionally, use the RETURNS clause to specify the output data type or length, as shown in the examples. The default is to publish to DATASET of maximum length using the only currently supported storage format, AVRO, so omit the RETURNS clause if that is your desired output.

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
7B2274797065223A226172726179222C226974656D73223A7B2274797065223A227265636F72642
22C226E616D65223A227265635F30222C226669656C6473223A5B7B226E616D65223A22656D704E
616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A22656D7044657074222
C2274797065223A22737472696E67227D5D7D7D0006001847656F72676520536D69746814416363
6F756E74696E67021C5061756C696E65204B72616D6572044852041853746576656E204D617A6
F16456E67696E656572696E6700
```

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

Example: Specifying Schemas

In this example, you specify schemas for the output format.

```
SELECT * FROM DATASET_PUBLISH
(
    ON (SELECT * FROM employeeTable)
    RETURNS (data DATASET STORAGE FORMAT AVRO)
        USING SCHEMA
        ('
```

```
{
    "type": "array",
    "items": {
        "type": "record",
        "name": "employeeRecord",
        "fields": [{
            "name": "empName",
            "type": "string"
        },
        {
            "name": "empDept",
            "type": "string"
        }]
    }
}
     ')
) AS avroFiles;

data
7B226E616D65223A22656D706C6F7965655461626C65222C2274797065223A227265636F7264222
C226669656C6473223A5B7B226E616D65223A22656D704964656E746966696572222C2274797065
223A22696E74227D2C7B226E616D65223A22656D7046756C6C4E616D65222C2274797065223A227
37472696E67227D2C7B226E616D65223A226465706172746D656E74222C2274797065223A227374
72696E67227D5D7D0006001847656F72676520536D697468144163636F756E74696E67021C50617
56C696E65204B72616D6572044852041853746576656E204D617A7A6F16456E67696E656572696E
6700
```

Example: Publishing Table Columns

In this example, 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
7B2274797065223A226172726179222C226974656D73223A7B2274797065223A227265636F72642
22C226E616D65223A227265635F30222C226669656C6473223A5B7B226E616D65223A22656D704E
616D65222C2274797065223A22737472696E67227D2C7B226E616D65223A22656D7044657074222
C2274797065223A22737472696E67227D5D7D7D0006001847656F72676520536D697468144163636
6F756E74696E67021C5061756C696E65204B72616D6572044852041853746576656E204D617A7A6
F16456E67696E656572696E6700
```

Example: Returning Results that are Not Aggregated

In the following 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
7B226E616D65223A22656D706C6F796655461626C65222C2274797065223A227265636F7264222
C226669656C6473223A5B7B226E616D65223A22656D704944222C2274797065223A22696E74227D
2C7B226E616D65223A22656D704E616D65222C2274797065223A22737472696E67227D2C7B226E6
16D65223A22656D7044657074222C2274797065223A22737472696E67227D5D7D00021847656F72
676520536D697468144163636F756E74696E67

7B226E616D65223A22656D706C6F796655461626C65222C2274797065223A227265636F7264222
C226669656C6473223A5B7B226E616D65223A22656D704944222C2274797065223A22696E74227D
2C7B226E616D65223A22656D704E616D65222C2274797065223A22737472696E67227D2C7B226E6
16D65223A22656D7044657074222C2274797065223A22737472696E67227D5D7D00041C5061756C
696E65204B72616D6572044852

7B226E616D65223A22656D706C6F796655461626C65222C2274797065223A227265636F7264222
C226669656C6473223A5B7B226E616D65223A22656D704944222C2274797065223A22696E74227D
2C7B226E616D65223A22656D704E616D65222C2274797065223A22737472696E67227D2C7B226E6
16D65223A22656D7044657074222C2274797065223A22737472696E67227D5D7D00061853746576
656E204D617A7A6F16456E67696E656572696E67
```

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 within Teradata. 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[*]

```
{                                       [{
    "type": "array",                        "empID": 5,
    "items": {                              "empName": "Kyle Newman",
        "type": "record",                   "empDept": "Engineering"
        "name": "rec_0",                }, {
        "fields": [{                        "empID": 3,
            "name": "record",               "empName": "Steven Mazzo",
            "type": {                       "empDept": "Engineering"
                "type": "record",       }, {
                "name":                     "empID": 1,
"rec_0_1448384735",                         "empName": "George Smith",
                "fields": [{                "empDept": "Accounting"
                    "name": "empID",    }, {
                    "type": "int"           "empID": 2,
                },                          "empName": "Pauline Kramer",
                {                           "empDept": "HR"
                    "name": "empName",  }, {
                    "type": "string"        "empID": 6,
                },                          "empName": "Pamela Giles",
                {                           "empDept": "Sales"
                    "name": "empDept",  }, {
                    "type": "string"        "empID": 4,
                }]
            }
        },
        {
            "name": "p",
```

```
            "type": "int"                    "empName": "Jose Hernandez",
        }]                                   "empDept": "Engineering"
    }                                    }]
}
```

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 Teradata 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.

CHAPTER 7

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 Teradata 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 of a Teradata table with a column-based schema.

The Teradata Database provides direct support for the files via 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 on a Teradata system 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 on a Teradata system 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 Teradata 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 into Teradata Database as BLOBs via BTEQ

The following example shows how to load Avro object container files into Teradata Database 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 columnn, 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 Teradata 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 Teradata 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 Teradata 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 Teradata 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).

APPENDIX A

Notation Conventions

Overview

Throughout this book, these conventions describe the SQL syntax and code:

- Syntax diagrams, used to describe SQL syntax form, including options.
- Square braces in the text, used to represent options. The indicated parentheses are required when you specify options.

  For example:

  DECIMAL [(n[,m])] means the decimal data type can be defined optionally:

  - without specifying the precision value n or scale value m specifying precision (n) only
  - specifying both values (n and m)
  - you cannot specify scale without first defining precision.
- CHARACTER [(n)] means that use of (n) is optional.

  The values for n and m are integers in all cases.
- Japanese character code shorthand notation, used to represent unprintable Japanese characters.

Related Topics

For more information about:

- Syntax diagrams, see Syntax Diagram Conventions.
- Japanese characters, see Character Shorthand Notation Used in This Book.

Syntax Diagram Conventions

Notation Conventions

Item	Definition and Comments
Letter	An uppercase or lowercase alphabetic character ranging from A through Z.
Number	A digit ranging from 0 through 9. Do not use commas when typing a number with more than 3 digits.
Word	Keywords and variables. • UPPERCASE LETTERS represent a keyword. Syntax diagrams show all keywords in uppercase, unless operating system restrictions require them to be in lowercase. • lowercase letters represent a keyword that you must type in lowercase, such as a Linux command.

Item	Definition and Comments
	• Mixed Case letters represent exceptions to uppercase and lowercase rules. The exceptions are noted in the syntax explanation. • *lowercase italic letters* represent a variable such as a column or table name. Substitute the variable with a proper value. • **lowercase bold letters** represent an excerpt from the diagram. The excerpt is defined immediately following the diagram that contains it. • UNDERLINED LETTERS represent the default value. This applies to both uppercase and lowercase words.
Spaces	Use one space between items such as keywords or variables.
Punctuation	Type all punctuation exactly as it appears in the diagram.

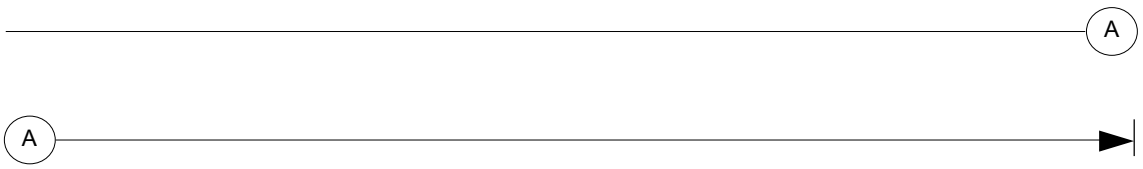
Paths

The main path along the syntax diagram begins at the left with a keyword, and proceeds, left to right, to the vertical bar, which marks the end of the diagram. Paths that do not have an arrow or a vertical bar only show portions of the syntax.

The only part of a path that reads from right to left is a loop.

Continuation Links

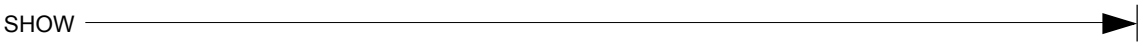
Paths that are too long for one line use continuation links. Continuation links are circled letters indicating the beginning and end of a link:



When you see a circled letter in a syntax diagram, go to the corresponding circled letter and continue reading.

Required Entries

Required entries appear on the main path:



If you can choose from more than one entry, the choices appear vertically, in a stack. The first entry appears on the main path:

Optional Entries

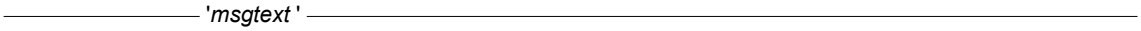
You may choose to include or disregard optional entries. Optional entries appear below the main path:

If you can optionally choose from more than one entry, all the choices appear below the main path:

Some commands and statements treat one of the optional choices as a default value. This value is UNDERLINED. It is presumed to be selected if you type the command or statement without specifying one of the options.

Strings

String literals appear in apostrophes:



Abbreviations

If a keyword or a reserved word has a valid abbreviation, the unabbreviated form always appears on the main path. The shortest valid abbreviation appears beneath.

In the above syntax, the following formats are valid:

SHOW CONTROLS
SHOW CONTROL

Loops

A loop is an entry or a group of entries that you can repeat one or more times. Syntax diagrams show loops as a return path above the main path, over the item or items that you can repeat:

Read loops from right to left.

The following conventions apply to loops:

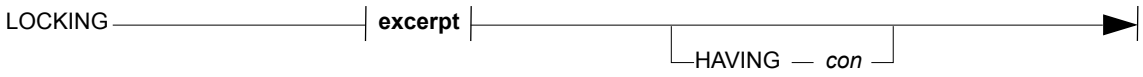
Item	Description	Example
maximum number of entries allowed	The number appears in a circle on the return path.	In the example, you may type *cname* a maximum of four times.
minimum number of entries allowed	The number appears in a square on the return path.	In the example, you must type at least three groups of column names.
separator character required between entries	The character appears on the return path. If the diagram does not show a separator character, use one blank space.	In the example, the separator character is a comma.

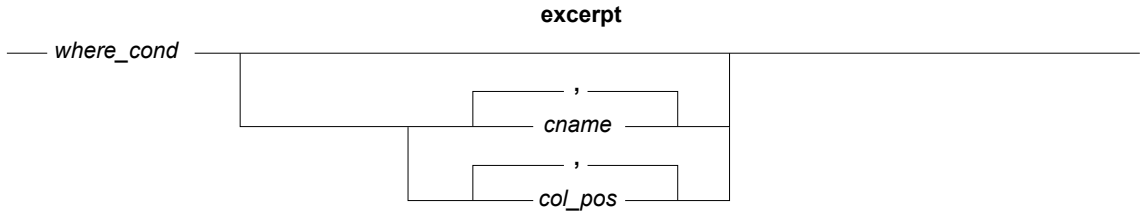
Item	Description	Example
delimiter character required around entries	The beginning and end characters appear outside the return path. Generally, a space is not needed between delimiter characters and entries.	In the example, the delimiter characters are the left and right parentheses.

Excerpts

Sometimes a piece of a syntax phrase is too large to fit into the diagram. Such a phrase is indicated by a break in the path, marked by (|) terminators on each side of the break. The name for the excerpted piece appears between the terminators in boldface type.

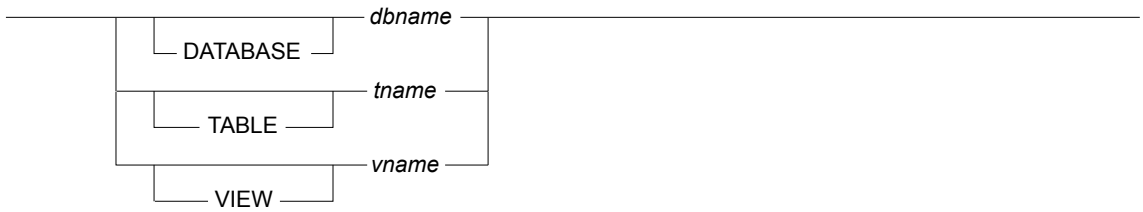
The boldface excerpt name and the excerpted phrase appears immediately after the main diagram. The excerpted phrase starts and ends with a plain horizontal line:





Multiple Legitimate Phrases

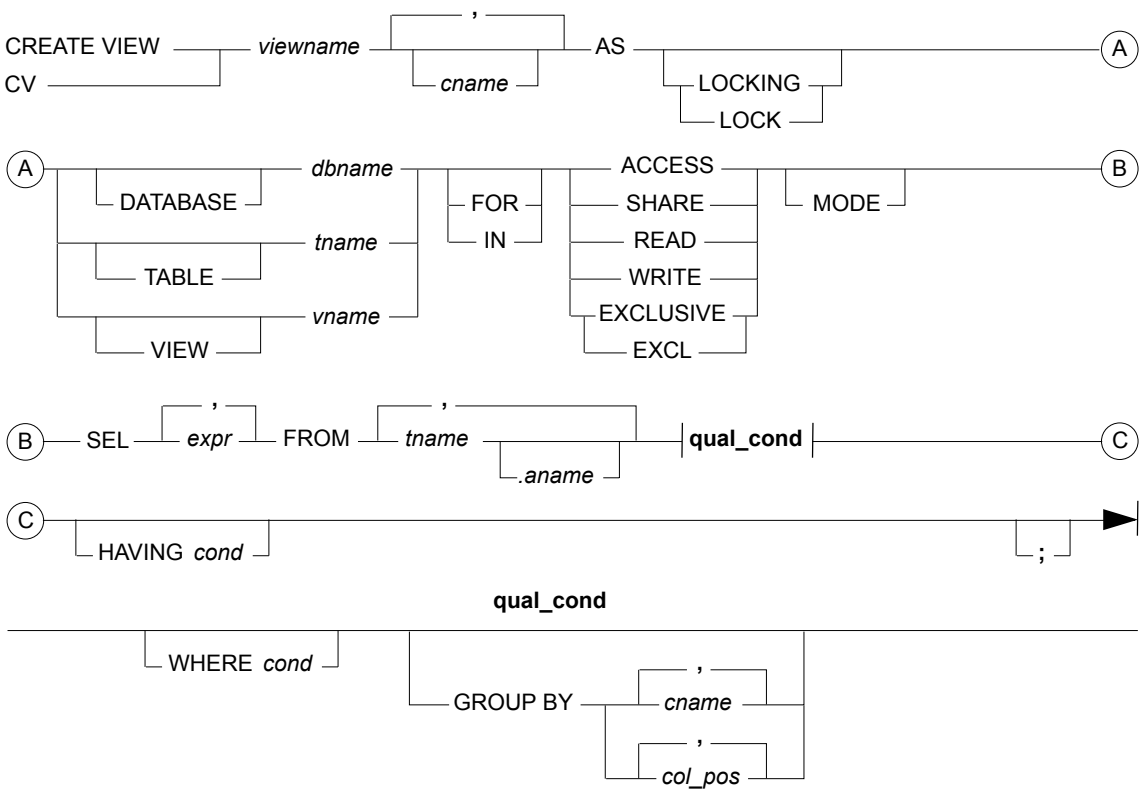
In a syntax diagram, it is possible for any number of phrases to be legitimate:



In this example, any of the following phrases are legitimate:

dbname
DATABASE *dbname*
tname
TABLE *tname*
vname
VIEW *vname*

Sample Syntax Diagram



Character Shorthand Notation Used in This Book

This book 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 book, 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 *International Character Set Support*.

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.
a-z	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
A-Z 0-9		
<	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 $_{ss3}$.
I	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by $_{ss2}$, forming an individual multibyte character.
Δ	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

APPENDIX B

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