Red Hat JBoss Data Virtualization 6.2

Development Guide Volume 3: Reference Material

This guide is intended for developers

Last Updated: 2017-10-20

Red Hat JBoss Data Virtualization 6.2 Development Guide Volume 3: Reference Material

This guide is intended for developers

Red Hat Customer Content Services

Legal Notice

Abstract

This document provides more information for developers creating custom solutions.

Table of Contents

CHAPTER 1. READ ME

1.1. BACK UP YOUR DATA

WARNING

Red Hat recommends that you back up your system settings and data before undertaking any of the configuration tasks mentioned in this book.

1.2. VARIABLE NAME: EAP_HOME

`EAP_HOME` refers to the root directory of the Red Hat JBoss Enterprise Application Platform installation on which JBoss Data Virtualization has been deployed.

1.3. VARIABLE NAME: MODE

`MODE` will either be **`standalone`** or **`domain`** depending on whether JBoss Data Virtualization is running in standalone or domain mode. Substitute one of these whenever you see **`MODE`** in a file path in this documentation. (You need to set this variable yourself, based on where the product has been installed in your directory structure.)

1.4. RED HAT DOCUMENTATION SITE

Red Hat's official documentation site is available at https://access.redhat.com/site/documentation/. There you will find the latest version of every book, including this one.

CHAPTER 2. ARCHITECTURE

2.1. TERMINOLOGY

- VM or Process - a JBoss EAP instance running JBoss Data Virtualization.
- Host - a machine that is "hosting" one or more VMs.
- Service - a subsystem running in a VM (often in many VMs) and providing a related set of functionality

In addition to these main components, the service platform provides a core set of services available to applications built on top of the service platform. These services are:

- Session - the Session service manages active session information.
- Buffer Manager - the Buffer Manager service provides access to data management for intermediate results. See Section 2.2.2, “Buffer Management”.
- Transaction - the Transaction service manages global, local, and request scoped transactions. See Section 6.1, “Transaction Support” for more information.

2.2. DATA MANAGEMENT

2.2.1. Cursoring and Batching

JBoss Data Virtualization cursors all results, regardless of whether they are from one source or many sources, and regardless of what type of processing (joins, unions, etc.) have been performed on the results.

JBoss Data Virtualization processes results in batches. A batch is a set of records. The number of rows in a batch is determined by the buffer system property processor-batch-size and is scaled based on the estimated memory footprint of the batch.

Client applications have no direct knowledge of batches or batch sizes, but rather specify fetch size. However the first batch, regardless of fetch size is always proactively returned to synchronous clients. Subsequent batches are returned based on client demand for the data. Pre-fetching is utilized at both the client and connector levels.

2.2.2. Buffer Management

The buffer manager manages memory for all result sets used in the query engine. That includes result sets read from a connection factory, result sets used temporarily during processing, and result sets prepared for a user. Each result set is referred to in the buffer manager as a tuple source.

When retrieving batches from the buffer manager, the size of a batch in bytes is estimated and then allocated against the maximum limit.

Memory Management

The buffer manager has two storage managers, these being a memory manager and a disk manager. The buffer manager maintains the state of all the batches and determines when batches must be moved from memory to disk.

Disk Management

Each tuple source has a dedicated file (named by the ID) on disk. This file will be created only if at least one batch for the tuple source had to be swapped to disk. This is a random access file. The connector batch size and processor batch size properties define how many rows can exist in a batch and thus define how granular the batches are when stored into the storage manager. Batches are always read and written from the storage manager together at once.

The disk storage manager has a cap on the maximum number of open files to prevent running out of file handles. In cases with heavy buffering, this can cause wait times while waiting for a file handle to become available (the default max open files is 64).

2.2.3. Cleanup

When a tuple source is no longer needed, it is removed from the buffer manager. The buffer manager will remove it from both the memory storage manager and the disk storage manager. The disk storage manager will delete the file. In addition, every tuple source is tagged with a "group name" which is typically the session ID of the client. When the client's session is terminated (by closing the connection, server detecting client shutdown, or administrative termination), a call is sent to the buffer manager to remove all tuple sources for the session.

In addition, when the query engine is shutdown, the buffer manager is shut down, which will remove all state from the disk storage manager and cause all files to be closed. When the query engine is stopped, it is safe to delete any files in the buffer directory as they are not used across query engine restarts and must be due to a system crash where buffer files were not cleaned up.

2.3. QUERY TERMINATION

2.3.1. Canceling Queries

When a query is canceled, processing will be stopped in the query engine and in all connectors involved in the query. The semantics of what a connector does in response to a cancellation command is dependent on the connector implementation. For example, JDBC connectors will asynchronously call cancel on the underlying JDBC driver, which may or may not actually support this method.

2.3.2. User Query Timeouts

User query timeouts in Data Virtualization can be managed on the client-side or server-side. Timeouts are only relevant for the first record returned. If the first record has not been received by the client within the specified timeout period, a "cancel" command is issued to the server for the request and no results are returned to the client. The cancel command is issued asynchronously by the JDBC API without the client's intervention.

The JDBC API uses the query timeout set by the **`java.sql.Statement.setQueryTimeout`** method. You can also set a default statement timeout via the connection property QUERYTIMEOUT. ODBC clients may also use QUERYTIMEOUT as an execution property via a set statement to control the default timeout setting. See *Red Hat JBoss Development Guide: Client Development*for more on connection/execution properties and set statements.

Server-side timeouts start when the query is received by the engine. The timeout will be canceled if the first result is sent back before the timeout has ended. See Section 9.2, “VDB Definition: The VDB Element” for more on setting the query-timeout VDB property. See the *Red Hat JBoss Administration Guide* for more information on setting the default query timeout for all queries.

2.4. PROCESSING

2.4.1. Join Algorithms

Nested loop does the most obvious processing - for every row in the outer source, it compares with every row in the inner source. Nested loop is only used when the join criteria has no equi-join predicates.

Merge join first sorts the input sources on the joined columns. You can then walk through each side in parallel (effectively one pass through each sorted source) and when you have a match, emit a row. In general, merge join is on the order of n+m rather than n*m in nested loop. Merge join is the default algorithm.

Using costing information the engine may also delay the decision to perform a full sort merge join. Based upon the actual row counts involved, the engine can choose to build an index of the smaller side (which will perform similarly to a hash join) or to only partially sort the larger side of the relation.

Joins involving equi-join predicates are also eligible to be made into dependent joins (see Section 14.7.3, “Dependent Joins”).

2.4.2. Sort-Based Algorithms

Sorting is used as the basis of the Sort (ORDER BY), Grouping (GROUP BY), and DupRemoval (SELECT DISTINCT) operations. The sort algorithm is a multi-pass merge-sort that does not ever require all of the result set to be in memory, yet uses the maximal amount of memory allowed by the buffer manager.

It consists of two phases. The first phase ("sort") will take an unsorted input stream and produce one or more sorted input streams. Each pass reads as much of the unsorted stream as possible, sorts it, and writes it back out as a new stream. Since the stream size may be bigger than that of the memory, it may be written out as many sorted streams.

The second phase ("merge") consists of a set of phases that grab the next batch from as many sorted input streams as will fit in memory. It then repeatedly grabs the next tuple in sorted order from each stream and outputs merged sorted batches to a new sorted stream. At completion of the pass, all input streams are dropped. Hence, each pass reduces the number of sorted streams. The last stream remaining is the final output.

CHAPTER 3. SQL SUPPORT

3.1. SQL SUPPORT

JBoss Data Virtualization supports SQL for issuing queries and for defining view transformations.

JBoss Data Virtualization provides nearly all of the functionality of SQL-92 DML. SQL-99 and later features have been added as required. The following does not attempt to cover SQL exhaustively, but rather highlights SQL's usage within JBoss Data Virtualization.

See the appendix for the SQL grammar accepted by JBoss Data Virtualization.

See Section 3.9.2, “Command Statement” for information on how SQL is used in virtual procedures and update procedures.

3.2. IDENTIFIERS

3.2.1. Identifiers

SQL commands contain references to tables and columns. These references are in the form of identifiers, which uniquely identify the tables and columns in the context of the command.

All queries are processed in the context of a virtual database (VDB). Because information can be federated across multiple sources, tables and columns must be scoped in some manner to avoid conflicts. This scoping is provided by schemas, which contain the information for each data source or set of views.

Fully qualified table and column names are of the following form, where the separate 'parts' of the identifier are delimited by periods.

- TABLE: <schema_name>.<table_spec>
- COLUMN: <schema_name>.<table_spec>.<column_name>

Syntax Rules:

- Identifiers can consist of alphanumeric characters, or the underscore (_) character, and must begin with an alphabetic character. Any Unicode character may be used in an identifier.
- Identifiers in double quotes can have any contents. The double quote character can be used if it is escaped with an additional double quote; for example, `"some "" id"`.
- Because different data sources organize tables in different ways (some prepending catalog or schema or user information) JBoss Data Virtualization allows table specification to be a dot delimited construct.

NOTE

When a table specification contains a dot, resolving will allow for the match of a partial name against any number of the end segments in the name. For example, a table with the fully qualified name **`vdbname."sourceschema.sourcetable"`** would match the partial name **`sourcetable`**.

- Columns, schemas, alias identifiers cannot contain a dot.
- Identifiers, even when quoted, are not case sensitive in JBoss Data Virtualization.

Some examples of valid fully qualified table identifiers are:

- MySchema.Portfolios
- "MySchema.Portfolios"
- MySchema.MyCatalog.dbo.Authors

Some examples of valid fully qualified column identifiers are:

- MySchema.Portfolios.portfolioID
- "MySchema.Portfolios"."portfolioID"
- MySchema.MyCatalog.dbo.Authors.lastName

Fully qualified identifiers can always be used in SQL commands. Partial or unqualified forms can also be used, as long as the resulting names are unambiguous in the context of the command. Different forms of qualification can be mixed in the same query.

3.2.2. Reserved Words

Reserved words in JBoss Data Virtualization include the standard SQL 2003 Foundation, SQL/MED, and SQL/XML reserved words, as well as JBoss Data Virtualization specific words such as BIGINTEGER, BIGDECIMAL, or MAKEDEP.

See Also:

- Section A.2, “Reserved Keywords”
- Section A.4, “Reserved Keywords For Future Use”

3.3. EXPRESSIONS

3.3.1. Expressions

Identifiers, literals, and functions can be combined into expressions. Expressions can be used almost anywhere in a query -- SELECT, FROM (if specifying join criteria), WHERE, GROUP BY, HAVING, or ORDER BY.

JBoss Data Virtualization supports the following types of expressions:

Column identifiers

Refer to Section 3.3.2, “Column Identifiers”.

Literals

Refer to Section 3.3.3, “Literals”.

Aggregate functions

Refer to Section 3.3.4, “Aggregate Functions”.

Window functions

Refer to Section 3.3.5, “Window Functions” .

Case and searched case

Refer to Section 3.3.8, “Case and Searched Case” .

Scalar subqueries

Refer to Section 3.3.9, “Scalar Subqueries” .

Parameter references

Refer to Section 3.3.10, “Parameter References”.

Criteria

Refer to Section 3.3.11, “Criteria”.

3.3.2. Column Identifiers

Column identifiers are used to specify the output columns in SELECT statements, the columns and their values for INSERT and UPDATE statements, and criteria used in WHERE and FROM clauses. They are also used in GROUP BY, HAVING, and ORDER BY clauses. The syntax for column identifiers is defined in Section 3.2.1, “Identifiers”.

3.3.3. Literals

Literal values represent fixed values. These can be any of the standard data types. See Section 4.1, “Supported Types”.

Syntax Rules:

- Integer values will be assigned an integral data type big enough to hold the value (integer, long, or biginteger).
- Floating point values will always be parsed as a double.
- The keyword 'null' is used to represent an absent or unknown value and is inherently untyped. In many cases, a null literal value will be assigned an implied type based on context. For example, in the function '5 + null', the null value will be assigned the type 'integer' to match the type of the value '5'. A null literal used in the SELECT clause of a query with no implied context will be assigned to type 'string'.

Some examples of simple literal values are:

- `'abc'`
- `'isn''t true'` - use an extra single tick to escape a tick in a string with single ticks
- `5`
- `-37.75e01` - scientific notation
- `100.0` - parsed as BigDecimal

- `true`
- `false`
- `'\u0027'` - unicode character

3.3.4. Aggregate Functions

Aggregate functions take sets of values from a group produced by an explicit or implicit GROUP BY and return a single scalar value computed from the group.

JBoss Data Virtualization supports the following aggregate functions:

- **`COUNT(*)`** - count the number of values (including nulls and duplicates) in a group
- **`COUNT(x)`** - count the number of values (excluding nulls) in a group
- **`SUM(x)`** - sum of the values (excluding nulls) in a group
- **`AVG(x)`** - average of the values (excluding nulls) in a group
- **`MIN(x)`** - minimum value in a group (excluding null)
- **`MAX(x)`** - maximum value in a group (excluding null)
- **`ANY(x)/SOME(x)`** - returns TRUE if any value in the group is TRUE (excluding null)
- **`EVERY(x)`** - returns TRUE if every value in the group is TRUE (excluding null)
- **`VAR_POP(x)`** - biased variance (excluding null) logically equals (sum(x^2) - sum(x)^2/count(x))/count(x); returns a double; null if count = 0
- **`VAR_SAMP(x)`** - sample variance (excluding null) logically equals (sum(x^2) - sum(x)^2/count(x))/(count(x) - 1); returns a double; null if count < 2
- **`STDDEV_POP(x)`** - standard deviation (excluding null) logically equals SQRT(VAR_POP(x))
- **`STDDEV_SAMP(x)`** - sample standard deviation (excluding null) logically equals SQRT(VAR_SAMP(x))
- **`TEXTAGG(FOR (expression [as name], ... [DELIMITER char] [QUOTE char] [HEADER] [ENCODING id] [ORDER BY ...])`** - CSV text aggregation of all expressions in each row of a group. When DELIMITER is not specified, by default comma (,) is used as delimiter. Double quotes(") is the default quote character. Use QUOTE to specify a different value. All non-null values will be quoted. If HEADER is specified, the result contains the header row as the first line. The header line will be present even if there are no rows in a group. This aggregation returns a BLOB. See Section 3.6.14, “ORDER BY Clause”. Example:

  ```
  TEXTAGG(col1, col2 as name DELIMITER '|' HEADER ORDER BY col1)
  ```

- **`XMLAGG(xml_expr [ORDER BY ...])`** - XML concatenation of all XML expressions in a group (excluding null). The ORDER BY clause cannot reference alias names or use positional ordering. See Section 3.6.14, “ORDER BY Clause”.

- **JSONARRAY_AGG(x [ORDER BY ...])** - creates a JSON array result as a CLOB including null value. The ORDER BY clause cannot reference alias names or use positional ordering. Also see Section 3.4.15, “JSON Functions”. Integer value example:

  ```
  jsonArray_Agg(col1 order by col1 nulls first)
  ```

  could return

  ```
  [null,null,1,2,3]
  ```

- **STRING_AGG(x, delim)** - creates a lob results from the concatenation of x using the delimiter delim. If either argument is null, no value is concatenated. Both arguments are expected to be character (string/clob) or binary (varbinary, blob) and the result will be clob or blob respectively. DISTINCT and ORDER BY are allowed in STRING_AGG. Example:

  ```
  string_agg(col1, ',' ORDER BY col1 ASC)
  ```

  could return

  ```
  'a,b,c'
  ```

- **agg([DISTINCT|ALL] arg ... [ORDER BY ...])** - a user defined aggregate function

Syntax Rules:

- Some aggregate functions may contain the keyword 'DISTINCT' before the expression, indicating that duplicate expression values should be ignored. DISTINCT is not allowed in COUNT(*) and is not meaningful in MIN or MAX (result would be unchanged), so it can be used in COUNT, SUM, and AVG.
- Aggregate functions cannot be used in FROM, GROUP BY, or WHERE clauses without an intervening query expression.
- Aggregate functions cannot be nested within another aggregate function without an intervening query expression.
- Aggregate functions may be nested inside other functions.
- Any aggregate function may take an optional FILTER clause of the following form:

  ```
  FILTER ( WHERE condition )
  ```

  The condition may be any boolean value expression that does not contain a subquery or a correlated variable. The filter will logically be evaluated for each row prior to the grouping operation. If false, the aggregate function will not accumulate a value for the given row.

- User defined aggregate functions need ALL specified if no other aggregate specific constructs are used to distinguish the function as an aggregate rather than normal function.

For more information on aggregates, refer to Section 3.6.12, “GROUP BY Clause” and Section 3.6.13, “HAVING Clause”.

3.3.5. Window Functions

JBoss Data Virtualization supports ANSI SQL 2003 window functions. A window function allows an aggregate function to be applied to a subset of the result set, without the need for a GROUP BY clause. A window function is similar to an aggregate function, but requires the use of an OVER clause or window specification.

Usage:

```
aggregate|ranking OVER ([PARTITION BY expression [, expression]*] [ORDER
BY ...])
```

In the above example, **`aggregate`** can be any of those in Section 3.3.4, “Aggregate Functions”. Ranking can be one of ROW_NUMBER(), RANK(), DENSE_RANK().

Syntax Rules:

- Window functions can only appear in the SELECT and ORDER BY clauses of a query expression.
- Window functions cannot be nested in one another.
- Partitioning and ORDER BY expressions cannot contain subqueries or outer references.
- The ranking (ROW_NUMBER, RANK, DENSE_RANK) functions require the use of the window specification ORDER BY clause.
- An XMLAGG ORDER BY clause cannot be used when windowed.
- The window specification ORDER BY clause cannot reference alias names or use positional ordering.
- Windowed aggregates may not use DISTINCT if the window specification is ordered.

3.3.6. Window Functions: Analytical Function Definitions

- ROW_NUMBER() - functionally the same as COUNT(*) with the same window specification. Assigns a number to each row in a partition starting at 1.
- RANK() - Assigns a number to each unique ordering value within each partition starting at 1, such that the next rank is equal to the count of prior rows.
- DENSE_RANK() - Assigns a number to each unique ordering value within each partition starting at 1, such that the next rank is sequential.

3.3.7. Window Functions: Processing

Window functions are logically processed just before creating the output from the SELECT clause. Window functions can use nested aggregates if a GROUP BY clause is present. There is no guaranteed effect on the output ordering from the presence of window functions. The SELECT statement must have an ORDER BY clause to have a predictable ordering.

JBoss Data Virtualization will process all window functions with the same window specification together. In general, a full pass over the row values coming into the SELECT clause will be required for each unique window specification. For each window specification the values will be grouped according to the PARTITION BY clause. If no PARTITION BY clause is specified, then the entire input is treated as a single partition. The output value is determined based upon the current row value, its peers (that is rows that are the same with respect to their ordering), and all prior row values based upon ordering in the partition. The ROW_NUMBER function will assign a unique value to every row regardless of the number of peers.

Example windowed results:

```
SELECT name, salary, max(salary) over (partition by name) as max_sal,
          rank() over (order by salary) as rank, dense_rank() over (order
by salary) as dense_rank,
          row_number() over (order by salary) as row_num FROM employees
```

name	salary	max_sal	rank	dense_rank	row_num
John	100000	100000	2	2	2
Henry	50000	100000	5	4	5
John	60000	60000	3	3	3
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

3.3.8. Case and Searched Case

JBoss Data Virtualization supports two forms of the CASE expression which allows conditional logic in a scalar expression.

Supported forms:

- CASE <expr> (WHEN <expr> THEN <expr>)+ [ELSE expr] END
- CASE (WHEN <criteria> THEN <expr>)+ [ELSE expr] END

Each form allows for an output based on conditional logic. The first form starts with an initial expression and evaluates WHEN expressions until the values match, and outputs the THEN expression. If no WHEN is matched, the ELSE expression is output. If no WHEN is matched and no ELSE is specified, a null literal value is output. The second form (the searched case expression) searches the WHEN clauses, which specify an arbitrary criteria to evaluate. If any criteria evaluates to true, the THEN expression is evaluated and output. If no WHEN is true, the ELSE is evaluated or NULL is output if none exists.

3.3.9. Scalar Subqueries

Subqueries can be used to produce a single scalar value in the SELECT, WHERE, or HAVING clauses only. A scalar subquery must have a single column in the SELECT clause and should return either 0 or 1 row. If no rows are returned, null will be returned as the scalar subquery value. For other types of subqueries, refer to Section 3.5.10, “Subqueries”.

3.3.10. Parameter References

Parameters are specified using a '?' symbol. Parameters may only be used with prepared statements or callable statements in JDBC. Each parameter is linked to a value specified by a one-based index in the JDBC API.

3.3.11. Criteria

Criteria may be:

- Predicates that evaluate to true or false
- Logical criteria that combines criteria (AND, OR, NOT)
- A value expression with type boolean

Usage:

-
  ```
  criteria AND|OR criteria
  ```

-
  ```
  NOT criteria
  ```

-
  ```
  (criteria)
  ```

-
  ```
  expression (=|<>|!=|<|>|<=|>=) (expression|((ANY|ALL|SOME)
  subquery))
  ```

-
  ```
  expression [NOT] IS NULL
  ```

-
  ```
  expression [NOT] IN (expression[,expression]*)|subquery
  ```

-
  ```
  expression [NOT] LIKE pattern [ESCAPE char]
  ```

  LIKE matches the string expression against the given string pattern. The pattern may contain % to match any number of characters and _ to match any single character. The escape character can be used to escape the match characters % and _.

-
  ```
  expression [NOT] SIMILAR TO pattern [ESCAPE char]
  ```

  SIMILAR TO is a cross between LIKE and standard regular expression syntax. % and _ are still used, rather than .* and . respectively.

  NOTE

  JBoss Data Virtualization does not exhaustively validate SIMILAR TO pattern values. Rather, the pattern is converted to an equivalent regular expression. Care should be taken not to rely on general regular expression features when using SIMILAR TO. If additional features are needed, then LIKE_REGEX should be used. Usage of a non-literal pattern is discouraged as pushdown support is limited.

-
  ```
  expression [NOT] LIKE_REGEX pattern
  ```

LIKE_REGEX allows for standard regular expression syntax to be used for matching. This differs from SIMILAR TO and LIKE in that the escape character is no longer used (\ is already the standard escape mechansim in regular expressions and % and _ have no special meaning. The runtime engine uses the JRE implementation of regular expressions - see the java.util.regex.Pattern class for details.

IMPORTANT

JBoss Data Virtualization does not exhaustively validate LIKE_REGEX pattern values. It is possible to use JRE only regular expression features that are not specified by the SQL specification. Additionally, not all sources support the same regular expression syntax or extensions. Care should be taken in pushdown situations to ensure that the pattern used will have the same meaning in JBoss Data Virtualization and across all applicable sources.

-
  ```
  EXISTS(subquery)
  ```

-
  ```
  expression [NOT] BETWEEN minExpression AND maxExpression
  ```

  JBoss Data Virtualization converts BETWEEN into the equivalent form expression >= minExpression AND expression <= maxExpression.

-
  ```
  expression
  ```

  Where expression has type boolean.

Syntax Rules:

- The precedence ordering from lowest to highest is: comparison, NOT, AND, OR.
- Criteria nested by parenthesis will be logically evaluated prior to evaluating the parent criteria.

Some examples of valid criteria are:

- (balance > 2500.0)
- 100*(50 - x)/(25 - y) > z
- concat(areaCode,concat('-',phone)) LIKE '314%1'

NOTE

Null values represent an unknown value. Comparison with a null value will evaluate to 'unknown', which can never be true even if 'not' is used.

3.3.12. Operator Precedence

JBoss Data Virtualization parses and evaluates operators with higher precedence before those with lower precedence. Operators with equal precedence are left associative. The following operator precedence is listed from highest to lowest:

Operator	Description
+,-	positive/negative value expression
*,/	multiplication/division
+,-	addition/subtraction
||	concat
criteria	see Section 3.3.11, “Criteria”

3.3.13. Criteria Precedence

JBoss Data Virtualization parses and evaluates conditions with higher precedence before those with lower precedence. Conditions with equal precedence are left associative. The following condition precedence is listed from highest to lowest:

Condition	Description
SQL operators	See Section 3.3.1, “Expressions”
EXISTS, LIKE, SIMILAR TO, LIKE_REGEX, BETWEEN, IN, IS NULL, <, <=, >, >=, =, <>	comparison
NOT	negation
AND	conjunction
OR	disjunction

Note however that to prevent lookaheads the parser does not accept all possible criteria sequences. For example "a = b is null" is not accepted, since by the left associative parsing we first recognize "a =", then look for a common value expression. "b is null" is not a valid common value expression. Thus nesting must be used, for example "(a = b) is null". See BNF for SQL Grammar for all parsing rules.

3.4. SCALAR FUNCTIONS

3.4.1. Scalar Functions

JBoss Data Virtualization provides an extensive set of built-in scalar functions. See Section 3.1, “SQL Support” and Section 4.1, “Supported Types”.

In addition, JBoss Data Virtualization provides the capability for user defined functions or UDFs. See *Red Hat JBoss Development Guide: Server Development*for adding UDFs. Once added, UDFs may be called like any other function.

3.4.2. Numeric Functions

Numeric functions return numeric values (integer, long, float, double, biginteger, bigdecimal). They generally take numeric values as inputs, though some take strings.

Table 3.1. Numeric Functions

Function	Definition	Data Type Constraint
+ - * /	Standard numeric operators	x in {integer, long, float, double, biginteger, bigdecimal}, return type is same as x **NOTE** The precision and scale of non-bigdecimal arithmetic function functions results matches that of Java. The results of bigdecimal operations match Java, except for division, which uses a preferred scale of max(16, dividend.scale + divisor.precision + 1), which then has trailing zeros removed by setting the scale to max(dividend.scale, normalized scale).
ABS(x)	Absolute value of x	See standard numeric operators above
ACOS(x)	Arc cosine of x	x in {double, bigdecimal}, return type is double
ASIN(x)	Arc sine of x	x in {double, bigdecimal}, return type is double
ATAN(x)	Arc tangent of x	x in {double, bigdecimal}, return type is double
ATAN2(x,y)	Arc tangent of x and y	x, y in {double, bigdecimal}, return type is double
CEILING(x)	Ceiling of x	x in {double, float}, return type is double
COS(x)	Cosine of x	x in {double, bigdecimal}, return type is double
COT(x)	Cotangent of x	x in {double, bigdecimal}, return type is double
DEGREES(x)	Convert x degrees to radians	x in {double, bigdecimal}, return type is double

Function	Definition	Data Type Constraint
EXP(x)	e^x	x in {double, float}, return type is double
FLOOR(x)	Floor of x	x in {double, float}, return type is double
FORMATBIGDECIMAL(x, y)	Formats x using format y	x is bigdecimal, y is string, returns string
FORMATBIGINTEGER(x, y)	Formats x using format y	x is biginteger, y is string, returns string
FORMATDOUBLE(x, y)	Formats x using format y	x is double, y is string, returns string
FORMATFLOAT(x, y)	Formats x using format y	x is float, y is string, returns string
FORMATINTEGER(x, y)	Formats x using format y	x is integer, y is string, returns string
FORMATLONG(x, y)	Formats x using format y	x is long, y is string, returns string
LOG(x)	Natural log of x (base e)	x in {double, float}, return type is double
LOG10(x)	Log of x (base 10)	x in {double, float}, return type is double
MOD(x, y)	Modulus (remainder of x / y)	x in {integer, long, float, double, biginteger, bigdecimal}, return type is same as x
PARSEBIGDECIMAL(x, y)	Parses x using format y	x, y are strings, returns bigdecimal
PARSEBIGINTEGER(x, y)	Parses x using format y	x, y are strings, returns biginteger
PARSEDOUBLE(x, y)	Parses x using format y	x, y are strings, returns double
PARSEFLOAT(x, y)	Parses x using format y	x, y are strings, returns float
PARSEINTEGER(x, y)	Parses x using format y	x, y are strings, returns integer
PARSELONG(x, y)	Parses x using format y	x, y are strings, returns long
PI()	Value of Pi	return is double
POWER(x,y)	x to the y power	x in {double, bigdecimal, biginteger}, return is the same type as x
RADIANS(x)	Convert x radians to degrees	x in {double, bigdecimal}, return type is double
RAND()	Returns a random number, using generator established so far in the query or initializing with system clock if necessary.	Returns double.

Function	Definition	Data Type Constraint
RAND(x)	Returns a random number, using new generator seeded with x.	x is integer, returns double.
ROUND(x,y)	Round x to y places; negative values of y indicate places to the left of the decimal point	x in {integer, float, double, bigdecimal} y is integer, return is same type as x
SIGN(x)	1 if x > 0, 0 if x = 0, -1 if x < 0	x in {integer, long, float, double, biginteger, bigdecimal}, return type is integer
SIN(x)	Sine value of x	x in {double, bigdecimal}, return type is double
SQRT(x)	Square root of x	x in {long, double, bigdecimal}, return type is double
TAN(x)	Tangent of x	x in {double, bigdecimal}, return type is double
BITAND(x, y)	Bitwise AND of x and y	x, y in {integer}, return type is integer
BITOR(x, y)	Bitwise OR of x and y	x, y in {integer}, return type is integer
BITXOR(x, y)	Bitwise XOR of x and y	x, y in {integer}, return type is integer
BITNOT(x)	Bitwise NOT of x	x in {integer}, return type is integer

3.4.3. Parsing Numeric Data Types from Strings

JBoss Data Virtualization provides a set of functions to parse formatted strings as various numeric data types:

- **`parseDouble`** - parses a string as a double
- **`parseFloat`** - parses a string as a float
- **`parseLong`** - parses a string as a long
- **`parseInteger`** - parses a string as an integer

For each function, you have to provide the formatting of the string. The formatting follows the convention established by the **`java.text.DecimalFormat`** class. See examples below.

Input String	Function Call to Format String	Output Value	Output Data Type

Input String	Function Call to Format String	Output Value	Output Data Type
'$25.30'	parseDouble(cost, '$#,##0.00;($#,##0.00)')	25.3	double
'25%'	parseFloat(percent, '#,##0%')	25	float
'2,534.1'	parseFloat(total, '#,##0.###;-#,##0.###')	2534.1	float
'1.234E3'	parseLong(amt, '0.###E0')	1234	long
'1,234,567'	parseInteger(total, '#,##0;-#,##0')	1234567	integer

NOTE

See http://download.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html for more information.

3.4.4. Formatting Numeric Data Types as Strings

JBoss Data Virtualization provides a set of functions to convert numeric data types into formatted strings:

- **`formatDouble`** - formats a double as a string
- **`formatFloat`** - formats a float as a string
- **`formatLong`** - formats a long as a string
- **`formatInteger`** - formats an integer as a string

For each function, you have to provide the formatting of the string. The formatting follows the convention established by the **`java.text.DecimalFormat`** class. See examples below.

Input Value	Input Data Type	Function Call to Format String	Output String
25.3	double	formatDouble(cost, '$#,##0.00;($#,##0.00)')	'$25.30'
25	float	formatFloat(percent, '#,##0%')	'25%'
2534.1	float	formatFloat(total, '#,##0.###;-#,##0.###')	'2,534.1'
1234	long	formatLong(amt, '0.###E0')	'1.234E3'
1234567	integer	formatInteger(total, '#,##0;-#,##0')	'1,234,567'

NOTE

See http://download.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html for more information.

3.4.5. String Functions

String functions generally take strings as inputs and return strings as outputs.

Unless specified, all of the arguments and return types in the following table are strings and all indexes are one-based. The zero index is considered to be before the start of the string.

IMPORTANT

Non-ASCII range characters or integers used by **`ASCII(x)`**, **`CHR(x)`**, and **`CHAR(x)`** may produce different results or exceptions depending on where the function is evaluated (JBoss Data Virtualization vs. source). JBoss Data Virtualization uses Java default int to char and char to int conversions, which operates over UTF16 values.

Table 3.2. String Functions

Function	Definition	DataType Constraint
x || y	Concatenation operator	x,y in {string}, return type is string
ASCII(x)	Provide ASCII value of the left most character in x. The empty string will return null.	return type is integer
CHR(x) CHAR(x)	Provide the character for ASCII value x	x in {integer}
CONCAT(x, y)	Concatenates x and y with ANSI semantics. If x and/or y is null, returns null.	x, y in {string}
CONCAT2(x, y)	Concatenates x and y with non-ANSI null semantics. If x and y is null, returns null. If only x or y is null, returns the other value.	x, y in {string}
ENDSWITH(x, y)	Checks if y ends with x. If only x or y is null, returns null.	x, y in {string}, returns boolean
INITCAP(x)	Make first letter of each word in string x capital and all others lowercase	x in {string}
INSERT(str1, start, length, str2)	Insert string2 into string1	str1 in {string}, start in {integer}, length in {integer}, str2 in {string}
LCASE(x)	Lowercase of x	x in {string}
LEFT(x, y)	Get left y characters of x	x in {string}, y in {integer}, return string

Function	Definition	DataType Constraint
LENGTH(x)	Length of x	return type is integer
LOCATE(x, y)	Find position of x in y starting at beginning of y	x in {string}, y in {string}, return integer
LOCATE(x, y, z)	Find position of x in y starting at z	x in {string}, y in {string}, z in {integer}, return integer
LPAD(x, y)	Pad input string x with spaces on the left to the length of y	x in {string}, y in {integer}, return string
LPAD(x, y, z)	Pad input string x on the left to the length of y using character z	x in {string}, y in {string}, z in {character}, return string
LTRIM(x)	Left trim x of blank characters	x in {string}, return string
QUERYSTRING(path [, expr [AS name] ...])	Returns a properly encoded query string appended to the given path. Null valued expressions are omitted, and a null path is treated as ''. Names are optional for column reference expressions. e.g. QUERYSTRING('path', 'value' as "&x", ' & ' as y, null as z) returns 'path?%26x=value&y=%20%26%20'	path, expr in {string}. name is an identifier
REPEAT(str1,instances)	Repeat string1 a specified number of times	str1 in {string}, instances in {integer} return string
REPLACE(x, y, z)	Replace all y in x with z	x,y,z in {string}, return string
RIGHT(x, y)	Get right y characters of x	x in {string}, y in {integer}, return string
RPAD(input string x, pad length y)	Pad input string x with spaces on the right to the length of y	x in {string}, y in {integer}, return string
RPAD(x, y, z)	Pad input string x on the right to the length of y using character z	x in {string}, y in {string}, z in {character}, return string
RTRIM(x)	Right trim x of blank characters	x is string, return string
SPACE(x)	Repeats space x times	x in {integer}

SUBSTRING(x, y)
 	
SUBSTRING(x FROM y)
	Get substring from x, from position y to the end of x	y in {integer}

Function	Definition	DataType Constraint

SUBSTRING(x, y, z)
 	
SUBSTRING(x FROM y FOR z)
	Get substring from x from position y with length z	y, z in {integer}

TO_CHARS(x, encoding)	Return a CLOB from the BLOB with the given encoding. BASE64, HEX, and the built-in Java Charset names are valid values for the encoding. **NOTE** For charsets, unmappable chars will be replaced with the charset default character. Binary formats, such as BASE64, will error in their conversion to bytes if an unrecognizable character is encountered.	x is a BLOB, encoding is a string, and returns a CLOB
TO_BYTES(x, encoding)	Return a BLOB from the CLOB with the given encoding. BASE64, HEX, and the builtin Java Charset names are valid values for the encoding.	x in a CLOB, encoding is a string, and returns a BLOB
TRANSLATE(x, y, z)	Translate string x by replacing each character in y with the character in z at the same position. Note that the second arg (y) and the third arg (z) must be the same length. If they are not equal, Red Hat JBoss data Virtualization throws this exception: 'TEIID30404 Source and destination character lists must be the same length.'	x in {string}
TRIM([[LEADING|TRAILING|BOTH] [x] FROM] y)	Trim character x from the leading, trailing, or both ends of string y. If LEADING/TRAILING/BOTH is not specified, BOTH is used by default. If no trim character x is specified, a blank space ' ' is used for x by default.	x in {character}, y in {string}
UCASE(x)	Uppercase of x	x in {string}

Function	Definition	DataType Constraint
UNESCAPE(x)	Unescaped version of x. Possible escape sequences are \b - backspace, \t - tab, \n - line feed, \f - form feed, \r - carriage return. \uXXXX, where X is a hex value, can be used to specify any unicode character. \XXX, where X is an octal digit, can be used to specify an octal byte value. If any other character appears after an escape character, that character will appear in the output and the escape character will be ignored.	x in {string}

3.4.6. Date/Time Functions

Date and time functions return or operate on dates, times, or timestamps.

Parse and format Date/Time functions use the convention established within the java.text.SimpleDateFormat class to define the formats you can use with these functions.

Table 3.3. Date and Time Functions

Function	Definition	Datatype Constraint
CURDATE()	Return current date	returns date
CURTIME()	Return current time	returns time
NOW()	Return current timestamp (date and time)	returns timestamp
DAYNAME(x)	Return name of day in the default locale	x in {date, timestamp}, returns string
DAYOFMONTH(x)	Return day of month	x in {date, timestamp}, returns integer
DAYOFWEEK(x)	Return day of week (Sunday=1, Saturday=7)	x in {date, timestamp}, returns integer
DAYOFYEAR(x)	Return day number	x in {date, timestamp}, returns integer
EXTRACT(YEAR|MONTH|DAY|HOUR|MINUTE|SECOND FROM x)	Return the given field value from the date value x. Produces the same result as the associated YEAR, MONTH, DAYOFMONTH, HOUR, MINUTE, SECOND functions. The SQL specification also allows for TIMEZONE_HOUR and TIMEZONE_MINUTE as extraction targets. In JBoss Data Virtualization, all date values are in the timezone of the server.	x in {date, time, timestamp}, returns integer

Function	Definition	Datatype Constraint
FORMATDATE(x, y)	Format date x using format y	x is date, y is string, returns string
FORMATTIME(x, y)	Format time x using format y	x is time, y is string, returns string
FORMATTIMESTAMP(x, y)	Format timestamp x using format y	x is timestamp, y is string, returns string
FROM_UNIXTIME (unix_timestamp)	Return the Unix timestamp (in seconds) as a Timestamp value	Unix timestamp (in seconds)
HOUR(x)	Return hour (in military 24-hour format)	x in {time, timestamp}, returns integer
MINUTE(x)	Return minute	x in {time, timestamp}, returns integer
MODIFYTIMEZONE (timestamp, startTimeZone, endTimeZone)	Returns a timestamp based upon the incoming timestamp adjusted for the differential between the start and end time zones. i.e. if the server is in GMT-6, then modifytimezone({ts '2006-01-10 04:00:00.0'},'GMT-7', 'GMT-8') will return the timestamp {ts '2006-01-10 05:00:00.0'} as read in GMT-6. The value has been adjusted 1 hour ahead to compensate for the difference between GMT-7 and GMT-8.	startTimeZone and endTimeZone are strings, returns a timestamp
MODIFYTIMEZONE (timestamp, endTimeZone)	Return a timestamp in the same manner as modifytimezone(timestamp, startTimeZone, endTimeZone), but will assume that the startTimeZone is the same as the server process.	Timestamp is a timestamp; endTimeZone is a string, returns a timestamp
MONTH(x)	Return month	x in {date, timestamp}, returns integer
MONTHNAME(x)	Return name of month in the default locale	x in {date, timestamp}, returns string
PARSEDATE(x, y)	Parse date from x using format y	x, y in {string}, returns date
PARSETIME(x, y)	Parse time from x using format y	x, y in {string}, returns time
PARSETIMESTAMP(x,y)	Parse timestamp from x using format y	x, y in {string}, returns timestamp
QUARTER(x)	Return quarter	x in {date, timestamp}, returns integer

Function	Definition	Datatype Constraint
SECOND(x)	Return seconds	x in {time, timestamp}, returns integer
TIMESTAMPCREATE(date, time)	Create a timestamp from a date and time	date in {date}, time in {time}, returns timestamp
TIMESTAMPADD(interval, count, timestamp)	Add a specified interval (hour, day of week, month) to the timestamp, where intervals can be: 1. SQL_TSI_FRAC_SECOND - fractional seconds (billionths of a second) 2. SQL_TSI_SECOND - seconds 3. SQL_TSI_MINUTE - minutes 4. SQL_TSI_HOUR - hours 5. SQL_TSI_DAY - days 6. SQL_TSI_WEEK - weeks where Sunday is the first day 7. SQL_TSI_MONTH - months 8. SQL_TSI_QUARTER - quarters (3 months), where the first quarter is months 1-3 9. SQL_TSI_YEAR - years **NOTE** The full interval amount based upon calendar fields will be added. For example adding 1 QUARTER will move the timestamp up by three full months and not just to the start of the next calendar quarter.	The interval constant may be specified either as a string literal or a constant value. Interval in {string}, count in {integer}, timestamp in {date, time, timestamp}

Function	Definition	Datatype Constraint
TIMESTAMPDIFF(interval, startTime, endTime)	Calculates the date part intervals crossed between the two timestamps. interval is one of the same keywords as those used for TIMESTAMPADD. If (endTime > startTime), a positive number will be returned. If (endTime < startTime), a negative number will be returned. The date part difference is counted regardless of how close the timestamps are. For example, '2000-01-02 00:00:00.0' is still considered 1 hour ahead of '2000-01-01 23:59:59.999999'. **NOTE** TIMESTAMPDIFF typically returns an integer, however JBoss Data Virtualization returns a long. You will encounter an exception if you expect a value out of the integer range from a pushed down TIMESTAMPDIFF. **NOTE** The implementation of TIMESTAMPDIFF in previous versions returned values based upon the number of whole canonical interval approximations (365 days in a year, 91 days in a quarter, 30 days in a month, etc.) crossed. For example the difference in months between 2013-03-24 and 2013-04-01 was 0, but based upon the date parts crossed is 1. See the System Properties section in *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for backwards compatibility.	Interval in {string}; startTime, endTime in {timestamp}, returns a long.
WEEK(x)	Return week in year (1-53). see also System Properties for customization.	x in {date, timestamp}, returns integer
YEAR(x)	Return four-digit year	x in {date, timestamp}, returns integer

3.4.7. Parsing Date Data Types from Strings

JBoss Data Virtualization does not implicitly convert strings that contain dates presented in different formats, such as '19970101' and '31/1/1996' to date-related data types. You can, however, use the following functions to explicitly convert strings with a different format to the appropriate data type:

- `parseDate`
- `parseTime`
- `parseTimestamp`

For each function, you have to provide the formatting of the string. The formatting follows the convention established by the `java.text.SimpleDateFormat` class. See examples below.

Table 3.4. Functions to Parse Dates

String	Function Call To Parse String
'19970101'	parseDate(myDateString, 'yyyyMMdd')
'31/1/1996'	parseDate(myDateString, 'dd"/"MM"/"yyyy')
'22:08:56 CST'	parseTime (myTime, 'HH:mm:ss z')
'03.24.2003 at 06:14:32'	parseTimestamp(myTimestamp, 'MM.dd.yyyy "at" hh:mm:ss')

NOTE

Formatted strings will be based on your default Java locale.

3.4.8. Specifying Time Zones

Time zones can be specified in several formats. Common abbreviations such as EST for "Eastern Standard Time" are allowed but discouraged, as they can be ambiguous. Unambiguous time zones are defined in the form continent or ocean/largest city. For example, America/New_York, America/Buenos_Aires, or Europe/London. Additionally, you can specify a custom time zone by GMT offset: GMT[+/-]HH:MM.

For example: GMT-05:00

3.4.9. Type Conversion Functions

Within your queries, you can convert between data types using the CONVERT or CAST keyword. Also see Section 4.2, “Type Conversions”.

Table 3.5. Type Conversion Functions

Function	Definition
CONVERT(x, type)	Convert x to type, where type is a JBoss Data Virtualization Base Type

Function	Definition
CAST(x AS type)	Convert x to type, where type is a JBoss Data Virtualization Base Type

These functions are identical other than syntax; CAST is the standard SQL syntax, CONVERT is the standard JDBC/ODBC syntax.

3.4.10. Choice Functions

Choice functions provide a way to select from two values based on some characteristic of one of the values.

Table 3.6. Type Conversion Functions

Function	Definition	Data Type Constraint
COALESCE(x,y +)	Returns the first non-null parameter	x and all y's can be any compatible types
IFNULL(x,y)	If x is null, return y; else return x	x, y, and the return type must be the same type but can be any type
NVL(x,y)	If x is null, return y; else return x	x, y, and the return type must be the same type but can be any type
NULLIF(param1 , param2)	Equivalent to case when (param1 = param2) then null else param1	param1 and param2 must be compatible comparable types

NOTE

IFNULL and NVL are aliases of each other. They are the same function.

3.4.11. Decode Functions

Decode functions allow you to have JBoss Data Virtualization examine the contents of a column in a result set and alter, or decode, the value so that your application can better use the results.

Table 3.7. Decode Functions

Function	Definition	Data Type Constraint

Function	Definition	Data Type Constraint
DECODESTRING(x, y [, z])	Decode column x using value pairs in y (with optional delimiter, z) and return the decoded column as a set of strings. **WARNING** Deprecated. Use a CASE expression instead.	All string
DECODEINTEGER(x, y [, z])	Decode column x using value pairs in y (with optional delimiter z) and return the decoded column as a set of integers. **WARNING** Deprecated. Use a CASE expression instead.	All string parameters, return integer

Within each function call, you include the following arguments:

1. x is the input value for the decode operation. This will generally be a column name.
2. y is the literal string that contains a delimited set of input values and output values.
3. z is an optional parameter on these methods that allows you to specify what delimiter the string specified in y uses.

For example, your application might query a table called PARTS that contains a column called IS_IN_STOCK which contains a Boolean value that you need to change into an integer for your application to process. In this case, you can use the DECODEINTEGER function to change the Boolean values to integers:

```
SELECT DECODEINTEGER(IS_IN_STOCK, 'false, 0, true, 1') FROM
PartsSupplier.PARTS;
```

When JBoss Data Virtualization encounters the value false in the result set, it replaces the value with 0.

If, instead of using integers, your application requires string values, you can use the DECODESTRING function to return the string values you need:

```
SELECT DECODESTRING(IS_IN_STOCK, 'false, no, true, yes, null') FROM
PartsSupplier.PARTS;
```

In addition to two input/output value pairs, this sample query provides a value to use if the column does not contain any of the preceding input values. If the row in the IS_IN_STOCK column does not contain true or false, JBoss Data Virtualization inserts a null into the result set.

When you use these DECODE functions, you can provide as many input/output value pairs as you would like within the string. By default, JBoss Data Virtualization expects a comma delimiter, but you can add a third parameter to the function call to specify a different delimiter:

```
SELECT DECODESTRING(IS_IN_STOCK, 'false:no:true:yes:null',':') FROM
PartsSupplier.PARTS;
```

You can use keyword null in the DECODE string as either an input value or an output value to represent a null value. However, if you need to use the literal string null as an input or output value (which means the word null appears in the column and not a null value) you can put the word in quotes: "null".

```
SELECT DECODESTRING( IS_IN_STOCK,
'null,no,"null",no,nil,no,false,no,true,yes' ) FROM PartsSupplier.PARTS;
```

If the DECODE function does not find a matching output value in the column and you have not specified a default value, the DECODE function will return the original value JBoss Data Virtualization found in that column.

3.4.12. Lookup Function

The Lookup function provides a way to speed up access to values in a lookup table (also known as a code table or reference table). The Lookup function caches all key and return column pairs specified in the function for the given table. Subsequent lookups against the same table using the same key and return columns will use the cached values. This caching accelerates response time to queries that use the lookup tables.

In the following example, based on the lookup table, **`codeTable`**, the following function will find the row where **`keyColumn`** has the value, **`keyValue`**, and return the associated **`returnColumn`** value (or null if no matching key is found).

```
LOOKUP(codeTable, returnColumn, keyColumn, keyValue)
```

`codeTable` must be a string literal that is the fully qualified name of the target table. **`returnColumn`** and **`keyColumn`** must also be string literals and match corresponding column names in **`codeTable`**. **`keyValue`** can be any expression that must match the datatype of the **`keyColumn`**. The return data type matches that of **`returnColumn`**.

Consider the following example in which the **`ISOCountryCodes`** table is used to translate country names to ISO codes:

```
lookup('ISOCountryCodes', 'CountryCode', 'CountryName', 'UnitedStates')
```

CountryName represents a key column and **CountryCode** represents the ISO code of the country. A query to this lookup table would provide a **CountryName**, in this case 'UnitedStates', and expect a **CountryCode** in response.

NOTE

JBoss Data Virtualization unloads these cached lookup tables when you stop and restart JBoss Data Virtualization. Thus, it is best not to use this function for data that is subject to updates or specific to a session or user (including row based security and column masking effects). It is best used for data that does not change over time. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more on the caching aspects of the lookup function.

IMPORTANT

- The key column must contain unique values. If the column contains duplicate values, an exception will be thrown.

3.4.13. System Functions

System functions provide access to information in JBoss Data Virtualization from within a query.

Function	Definition	Data Type Constraint
COMMANDPAYLOAD([key])	If the key parameter is provided, the command payload object is cast to a java.util.Properties object and the corresponding property value for the key is returned. If the key is not specified, the return value is the command payload toString value. The command payload is set by the **TeiidStatement.setPayload** method on the Data Virtualization JDBC API extensions on a per-query basis.	key in {string}, return value is string

Function	Definition	Data Type Constraint
`ENV(key)`	Retrieve a system environment property. **NOTE** The only key specific to the current session is 'sessionid'. The preferred mechanism for getting the session id is with the session_id() function. **NOTE** To prevent untrusted access to system properties, this function is not enabled by default. The ENV function may be enabled via the allowEnvFunction property.	key in {string}, return value is string
`SESSION_ID()`	Retrieve the string form of the current session id.	return value is string
`USER()`	Retrieve the name of the user executing the query.	return value is string
`CURRENT_DATABASE()`	Retrieve the catalog name of the database which, for the VDB, is the VDB name.	return value is string
`TEIID_SESSION_GET(name)`	Retrieve the session variable. A null name will return a null value. Typically you will use the a get wrapped in a CAST to convert to the desired type.	name in {string}, return value is object
`TEIID_SESSION_SET(name, value)`	Set the session variable. The previous value for the key or null will be returned. A set has no effect on the current transaction and is not affected by commit/rollback.	name in {string}, value in {object}, return value is object

3.4.14. XML Functions

XML functions provide functionality for working with XML data.

```
 TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    CustomerName varchar(25),
    ContactName varchar(25)
    Address varchar(50),
    City varchar(25),
    PostalCode varchar(25),
    Country varchar(25),
);
```

use this data

```
CustomerID  CustomerName  ContactName  Address  City  PostalCode  Country
87  Wartian Herkku  Pirkko Koskitalo  Torikatu 38  Oulu  90110  Finland
88  Wellington Importadora  Paula Parente  Rua do Mercado, 12  Resende
08737-363  Brazil
89  White Clover Markets  Karl Jablonski  305 - 14th Ave. S. Suite 3B
Seattle  98128  USA
```

XMLCAST

Cast to or from XML:

XMLCAST(expression AS type)

Expression or type must be XML. The return value will be typed as type. This is the same functionality as XMLTABLE uses to convert values to the desired runtime type - with the exception that array type targets are not supported with XMLCAST.

XMLCOMMENT

```
XMLCOMMENT(comment)
```

Returns an XML comment.

comment is a string. Return value is XML.

XMLCONCAT

```
XMLCONCAT(content [, content]*)
```

Returns XML with the concatenation of the given XML types. If a value is null, it will be ignored. If all values are null, null is returned. This is how you concatenate two or more XML fragments:

```
     SELECT XMLCONCAT( XMLELEMENT("name", CustomerName),
XMLPARSE(CONTENT '
     <a>
      b
     </a>' WELLFORMED) ) FROM Customer c WHERE c.CustomerID = 87;
==========================================================
     <name>
      Wartian Herkku
     </name>
     <a>
      b
     </a>

```

content is XML. Return value is XML.

XMLELEMENT

```
XMLELEMENT([NAME] name [, <NSP>] [, <ATTR>][, content]*)
ATTR:=XMLATTRIBUTES(exp [AS name] [, exp [AS name]]*)
NSP:=XMLNAMESPACES((uri AS prefix | DEFAULT uri | NO DEFAULT))+
```

Returns an XML element with the given name and content. If the content value is of a type other than XML, it will be escaped when added to the parent element. Null content values are ignored. Whitespace in XML or the string values of the content is preserved, but no whitespace is added between content values.

XMLNAMESPACES is used to provide namespace information. NO DEFAULT is equivalent to defining the default namespace to the null URI - **xmlns=""**. Only one DEFAULT or NO DEFAULT namespace item may be specified. The namespace prefixes **xmlns** and **xml** are reserved.

If an attribute name is not supplied, the expression must be a column reference, in which case the attribute name will be the column name. Null attribute values are ignored.

For example, with an xml_value of <doc/>,

```
XMLELEMENT(NAME "elem", 1, '<2/>', xml_value)
```

returns

```
<elem>1&lt;2/&gt;<doc/><elem/>
```

name and **prefix** are identifiers. **uri** is a string literal. **content** can be any type. Return value is XML. The return value is valid for use in places where a document is expected.

```
    SELECT XMLELEMENT("name", CustomerName)
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<name>Wartian Herkku</name>
"Multiple Columns"
SELECT XMLELEMENT("customer",
          XMLELEMENT("name", c.CustomerName),
          XMLELEMENT("contact", c.ContactName))
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact>
```

```
</customer>
"Columns as Attributes"
SELECT XMLELEMENT("customer",
          XMLELEMENT("name", c.CustomerName,
            XMLATTRIBUTES(
                  "contact" as c.ContactName,
                  "id" as c.CustomerID
            )
          )
       )
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<customer><name contact="Pirkko Koskitalo" id="87">Wartian Herkku</name>
</customer>
```

XMLFOREST

```
XMLFOREST(content [AS name] [, <NSP>] [, content [AS name]]*)
```

Returns an concatenation of XML elements for each content item. See XMLELEMENT for the definition of NSP. If a name is not supplied for a content item, the expression must be a column reference, in which case the element name will be a partially escaped version of the column name.

name is an identifier. **`content`** can be any type. Return value is XML.

You can use XMLFORREST to simplify the declaration of multiple XMLELEMENTS, XMLFOREST function allows you to process multiple columns at once:

```
SELECT XMLELEMENT("customer",
          XMLFOREST(
             c.CustomerName AS "name",
             c.ContactName AS "contact"
          ))
FROM   Customer c
WHERE  c.CustomerID = 87;

==========================================================
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact>
</customer>
XMLAGG

XMLAGG is an aggregate function, that takes a collection of XML elements
and returns an aggregated XML document.
XMLAGG(xml)

From above example in XMLElement, each row in the Customer table table
will generate row of XML if there are multiple rows matching the
criteria. That will generate a valid XML, but it will not be well
formed, because it lacks the root element. XMLAGG can used to correct
that
"Example"
SELECT XMLELEMENT("customers",
         XMLAGG(
           XMLELEMENT("customer",
             XMLFOREST(
               c.CustomerName AS "name",
               c.ContactName AS "contact"
             )))
FROM   Customer c


==========================================================
<customers>
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact>
</customer>
<customer><name>Wellington Importadora</name><contact>Paula
Parente</contact></customer>
<customer><name>White Clover Markets</name><contact>Karl
Jablonski</contact></customer>
</customers>
```

XMLPARSE

```
XMLPARSE((DOCUMENT|CONTENT) expr [WELLFORMED])
```

Returns an XML type representation of the string value expression. If DOCUMENT is specified, then the expression must have a single root element and may or may not contain an XML declaration. If WELLFORMED is specified then validation is skipped; this is especially useful for CLOB and BLOB known to already be valid.

expr in {string, clob, blob and varbinary}. Return value is XML.

If DOCUMENT is specified then the expression must have a single root element and may or may not contain an XML declaration. If WELLFORMED is specified then validation is skipped; this is especially useful for CLOB and BLOB known to already be valid.

```
SELECT XMLPARSE(CONTENT '<customer><name>Wartian Herkku</name>
<contact>Pirkko Koskitalo</contact></customer>' WELLFORMED);
```

Will return a SQLXML with contents:

```
<customer><name>Wartian Herkku</name><contact>Pirkko Koskitalo</contact>
</customer>
```

XMLPI

```
XMLPI([NAME] name [, content])
```

Returns an XML processing instruction.

name is an identifier. **content** is a string. Return value is XML.

XMLQUERY

```
XMLQUERY([<NSP>] xquery [<PASSING>] [(NULL|EMPTY) ON EMPTY]]
PASSING:=PASSING exp [AS name] [, exp [AS name]]*
```

Returns the XML result from evaluating the given **xquery**. See XMLELEMENT for the definition of NSP. Namespaces may also be directly declared in the XQuery prolog.

The optional PASSING clause is used to provide the context item, which does not have a name, and named global variable values. If the XQuery uses a context item and none is provided, then an exception will be raised. Only one context item may be specified and should be an XML type. All non-context non-XML passing values will be converted to an appropriate XML type.

The ON EMPTY clause is used to specify the result when the evaluated sequence is empty. EMPTY ON EMPTY, the default, returns an empty XML result. NULL ON EMPTY returns a null result.

xquery in string. Return value is XML.

NOTE

XMLQUERY is part of the SQL/XML 2006 specification.

See also XMLTABLE.

XMLEXISTS

Returns true if a non-empty sequence would be returned by evaluating the given xquery.

```
XMLEXISTS([<NSP>] xquery [<PASSING>]]

PASSING:=PASSING exp [AS name] [, exp [AS name]]*
```

Namespaces may also be directly declared in the xquery prolog.

The optional PASSING clause is used to provide the context item, which does not have a name, and named global variable values. If the xquery uses a context item and none is provided, then an exception will be raised. Only one context item may be specified and should be an XML type. All non-context non-XML passing values will be converted to an appropriate XML type. Null/Unknown will be returned if the context item evaluates to null.

xquery in string. Return value is boolean.

XMLEXISTS is part of the SQL/XML 2006 specification.

XMLSERIALIZE

```
XMLSERIALIZE([(DOCUMENT|CONTENT)] xml [AS datatype] [ENCODING enc]
[VERSION ver] [(INCLUDING|EXCLUDING) XMLDECLARATION])
```

Returns a character type representation of the XML expression.

datatype may be character (string, varchar, clob) or binary (blob, varbinary). CONTENT is the default. If DOCUMENT is specified and the XML is not a valid document or fragment, then an exception is raised.

Return value matches data type. If no data type is specified, then CLOB will be assumed.

The encoding **enc** is specified as an identifier. A character serialization may not specify an encoding. The version **ver** is specified as a string literal. If a particular XMLDECLARATION is not specified, then the result will have a declaration only if performing a non UTF-8/UTF-16 or non version 1.0 document serialization or the underlying XML has an declaration. If CONTENT is being serialized, then the declaration will be omitted if the value is not a document or element.

The following example produces a BLOB of XML in UTF-16 including the appropriate byte order mark of FE FF and XML declaration:

```
XMLSERIALIZE(DOCUMENT value AS BLOB ENCODING "UTF-16" INCLUDING
XMLDECLARATION)
```

XSLTRANSFORM

```
XSLTRANSFORM(doc, xsl)
```

Applies an XSL stylesheet to the given document.

doc and **xsl** in {string, clob, xml}. Return value is a CLOB. If either argument is null, the result is null.

XPATHVALUE

```
XPATHVALUE(doc, xpath)
```

Applies the XPATH expression to the document and returns a string value for the first matching result. For more control over the results and XQuery, use the XMLQUERY function.

Matching a non-text node will still produce a string result, which includes all descendant text nodes.

doc and **xpath** in {string, clob, xml}. Return value is a string.

When the input document utilizes namespaces, it is sometimes necessary to specify XPATH that ignores namespaces. For example, given the following XML,

```
<?xml version="1.0" ?>
  <ns1:return xmlns:ns1="http://com.test.ws/exampleWebService">Hello<x>
World</x></return>
```

the following function results in 'Hello World'.

```
xpathValue(value, '/*[local-name()="return"])
```

3.4.15. JSON Functions

JSON functions provide functionality for working with JSON (JavaScript Object Notation) data.

JSONTOXML

```
JSONTOXML(rootElementName, json)
```

Returns an XML document from JSON. The appropriate UTF encoding (8, 16LE. 16BE, 32LE, 32BE) will be detected for JSON BLOBS. If another encoding is used, see the TO_CHARS function (see Section 3.4.5, “String Functions”).

`rootElementName` is a string, `json` is in {clob, blob}. Return value is XML. The result is always a well-formed XML document.

The mapping to XML uses the following rules:

- The current element name is initially the `rootElementName`, and becomes the object value name as the JSON structure is traversed.
- All element names must be valid XML 1.1 names. Invalid names are fully escaped according to the SQLXML specification.
- Each object or primitive value will be enclosed in an element with the current name.
- Unless an array value is the root, it will not be enclosed in an additional element.
- Null values will be represented by an empty element with the attribute `xsi:nil="true"`
- Boolean and numerical value elements will have the attribute `xsi:type` set to `boolean` and `decimal` respectively.

Example 3.1. Sample JSON to XML for jsonToXml('person', x)

JSON:

```
{ "firstName" : "John" , "children" : [ "Randy", "Judy" ] }
```

XML:

```
<?xml version="1.0" ?
><person><firstName>John</firstName><children>Randy</children><childr
en>Judy</children></person>
```

Example 3.2. Sample JSON to XML for jsonToXml('person', x) with a root array.

JSON:

```
[{ "firstName" : "George" }, { "firstName" : "Jerry" }]
```

XML (Notice there is an extra "person" wrapping element to keep the XML well-formed):

```
<?xml version="1.0" ?
><person><person><firstName>George</firstName></person><person><first
Name>Jerry</firstName></person></person>
```

JSON:

Example 3.3. Sample JSON to XML for jsonToXml('root', x) with an invalid name.

```
{"/invalid" : "abc" }
```

XML:

Example 3.4. Sample JSON to XML for jsonToXml('root', x) with an invalid name.

```
<?xml version="1.0" ?>
<root>
  <_u002F_invalid>abc</_u002F_invalid>
</root>
```

JSONARRAY

```
JSONARRAY(value...)
```

Returns a JSON array.

value is any object convertable to a JSON value (see Section 3.4.16, “Conversion to JSON”). Return value is a CLOB marked as being valid JSON. Null values will be included in the result as null literals.

For example:

```
jsonArray('a"b', 1, null, false, {d'2010-11-21'})
```

returns

```
["a\"b",1,null,false,"2010-11-21"]
```

JSONOBJECT

```
JSONARRAY(value [as name] ...)
```

Returns a JSON object.

value is any object convertable to a JSON value (see Section 3.4.16, “Conversion to JSON”). Return value is a clob marked as being valid JSON.

Null values will be included in the result as null literals.

If a name is not supplied and the expression is a column reference, the column name will be used otherwise exprN will be used where N is the 1-based index of the value in the JSONARRAY expression.

For example:

```
jsonObject('a"b' as val, 1, null as "null")
```

returns

```
{"val":"a\"b","expr2":1,"null":null}
```

JSONPARSE

```
JSONPARSE(value, wellformed)
```

Validates and returns a JSON result.

`value` is blob with an appropriate JSON binary encoding (UTF-8, UTF-16, or UTF-32) or clob. **`wellformed`** is a boolean indicating that validation should be skipped. Return value is a CLOB marked as being valid JSON.

A null for either input will return null.

```
jsonParse('"a"')
```

3.4.16. Conversion to JSON

A straightforward specification compliant conversion is used for converting values into their appropriate JSON document form.

- null values are included as the null literal.
- values parsed as JSON or returned from a JSON construction function (JSONPARSE, JSONARRAY, JSONARRAY_AGG) will be directly appended into a JSON result.
- boolean values are included as true/false literals
- numeric values are included as their default string conversion - in some circumstances if not a number or +-infinity results are allowed, invalid JSON may be obtained.
- string values are included in their escaped/quoted form.
- binary values are not implicitly convertible to JSON values and require a specific prior to inclusion in JSON.
- all other values will be included as their string conversion in the appropriate escaped/quoted form.

3.4.17. Security Functions

Security functions provide the ability to interact with the security system.

HASROLE

```
hasRole([roleType,] roleName)
```

Whether the current caller has the JBoss Data Virtualization data role **`roleName`**.

`roleName` must be a string, the return type is boolean.

The two argument form is provided for backwards compatibility. **`roleType`** is a string and must be 'data'.

Role names are case-sensitive and only match JBoss Data Virtualization data roles (see Section 7.1, “Data Roles”). JAAS roles/groups names are not valid for this function, unless there is corresponding data role with the same name.

3.4.18. Miscellaneous Functions

array_get

```
array_get(array, index)
```

Returns the object value at a given array index.

`array` is the object type, **`index`** must be an integer, and the return type is object.

One-based indexing is used. The actual array value must be a **`java.sql.Array`** or Java array type. An exception will be thrown if the array value is the wrong type of the index is out of bounds.

array_length

```
array_length(array)
```

Returns the length for a given array.

`array` is the object type, and the return type is integer.

The actual array value must be a **`java.sql.Array`** or Java array type. An exception will be thrown if the array value is the wrong type.

uuid

```
uuid()
```

Returns a universally unique identifier.

The return type is string.

Generates a type 4 (pseudo randomly generated) UUID using a cryptographically strong random number generator. The format is XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX where each X is a hex digit.

3.4.19. Nondeterministic Function Handling

JBoss Data Virtualization categorizes functions by varying degrees of determinism. When a function is evaluated and to what extent the result can be cached are based upon its determinism level.

1. Deterministic - the function will always return the same result for the given inputs. Deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. Some functions, such as the **`lookup`** function, are not truly deterministic, but is treated as such for performance. All functions not categorized below are considered deterministic.

2. User Deterministic - the function will return the same result for the given inputs for the same

user. This includes the **hasRole** and user functions. User deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. If a user deterministic function is evaluated during the creation of a prepared processing plan, then the resulting plan will be cached only for the user.

3. Session Deterministic - the function will return the same result for the given inputs under the same user session. This category includes the **env** function. Session deterministic functions are evaluated by the engine as soon as all input values are known, which may occur as soon as the rewrite phase. If a session deterministic function is evaluated during the creation of a prepared processing plan, then the resulting plan will be cached only for the user's session.

4. Command Deterministic - the result of function evaluation is only deterministic within the scope of the user command. This category include the **curdate**, **curtime**, **now**, and **commandpayload** functions. Command deterministic functions are delayed in evaluation until processing to ensure that even prepared plans utilizing these functions will be executed with relevant values. Command deterministic function evaluation will occur prior to pushdown; however, multiple occurrences of the same command deterministic time function are not guaranteed to evaluate to the same value.

5. Nondeterministic - the result of function evaluation is fully nondeterministic. This category includes the **rand** function and UDFs marked as nondeterministic. Nondeterministic functions are delayed in evaluation until processing with a preference for pushdown. If the function is not pushed down, then it may be evaluated for every row in its execution context (for example, if the function is used in the select clause).

3.5. DML COMMANDS

3.5.1. DML Commands

JBoss Data Virtualization supports SQL for issuing queries and defining view transformations; see also Section 3.9.1, “Procedural Language” and Section 3.10.1, “Virtual Procedures” for how SQL is used in virtual procedures and update procedures. Nearly all these features follow standard SQL syntax and functionality, so any SQL reference can be used for more information.

There are 4 basic commands for manipulating data in SQL, corresponding to the standard create, read, update and delete (CRUD) operations: INSERT, SELECT, UPDATE, and DELETE. A MERGE statement acts as a combination of INSERT and UPDATE. In addition, procedures can be executed using the EXECUTE command or through a procedural relational command. See Section 3.5.8, “Procedural Relational Command”.

3.5.2. SELECT Command

The SELECT command is used to retrieve records for any number of relations.

A SELECT command consists of several clauses:

- **[WITH ...]**
- **SELECT ...**
- **[FROM ...]**
- **[WHERE ...]**
- **[GROUP BY ...]**

- `[HAVING ...]`
- `[ORDER BY ...]`
- `[(LIMIT ...) | ([OFFSET ...] [FETCH ...])]`
- `[OPTION ...]`

See Section 3.6.1, “DML Clauses” for more information about all of these clauses.

All of these clauses other than OPTION are defined by the SQL specification. The specification also specifies the order that these clauses will be logically processed. Below is the processing order where each stage passes a set of rows to the following stage. Note that this processing model is logical and does not represent the way any actual database engine performs the processing, although it is a useful model for understanding questions about SQL.

- WITH stage - gathers all rows from all WITH items in the order listed. Subsequent WITH items and the main query can reference a WITH item as if it is a table.
- FROM stage - gathers all rows from all tables involved in the query and logically joins them with a Cartesian product, producing a single large table with all columns from all tables. Joins and join criteria are then applied to filter rows that do not match the join structure.
- WHERE stage - applies a criteria to every output row from the FROM stage, further reducing the number of rows.
- GROUP BY stage - groups sets of rows with matching values in the GROUP BY columns.
- HAVING stage - applies criteria to each group of rows. Criteria can only be applied to columns that will have constant values within a group (those in the grouping columns or aggregate functions applied across the group).
- SELECT stage - specifies the column expressions that should be returned from the query. Expressions are evaluated, including aggregate functions based on the groups of rows, which will no longer exist after this point. The output columns are named using either column aliases or an implicit name determined by the engine. If SELECT DISTINCT is specified, duplicate removal will be performed on the rows being returned from the SELECT stage.
- ORDER BY stage - sorts the rows returned from the SELECT stage as desired. Supports sorting on multiple columns in specified order, ascending or descending. The output columns will be identical to those columns returned from the SELECT stage and will have the same name.
- LIMIT stage - returns only the specified rows (with skip and limit values).

This model helps to understand how SQL works. For example, columns aliased in the SELECT clause can only be referenced by alias in the ORDER BY clause. Without knowledge of the processing model, this can be somewhat confusing. Seen in light of the model, it is clear that the ORDER BY stage is the only stage occurring after the SELECT stage, which is where the columns are named. Because the WHERE clause is processed before the SELECT, the columns have not yet been named and the aliases are not yet known.

NOTE

The explicit table syntax **`TABLE x`** may be used as a shortcut for **`SELECT * FROM x`**.

3.5.3. INSERT Command

The INSERT command is used to add a record to a table.

Example Syntax

- INSERT INTO table (column,...) VALUES (value,...)
- INSERT INTO table (column,...) query

3.5.4. UPDATE Command

The UPDATE command is used to modify records in a table. The operation will result in 1 or more records being updated, or in no records being updated if none match the criteria.

Example Syntax

- UPDATE table SET (column=value,...) [WHERE criteria]

3.5.5. DELETE Command

The DELETE command is used to remove records from a table. The operation will result in 1 or more records being deleted, or in no records being deleted if none match the criteria.

Example Syntax

- DELETE FROM table [WHERE criteria]

3.5.6. MERGE Command

The MERGE command, also known as UPSERT, is used to add and/or update records. The JBoss Data Virtualization (non-ANSI) MERGE is simply a modified INSERT statement that requires the target table to have a primary key and for the target columns to cover the primary key. The MERGE operation will then check the existence of each row prior to INSERT and instead perform an UPDATE if the row already exists.

Example Syntax

-
  ```
  MERGE INTO table (column,...) VALUES (value,...)
  ```

-
  ```
  MERGE INTO table (column,...) query
  ```

NOTE

The MERGE statement is not currently pushed to sources, but rather will be broken down into the respective insert/update operations.

3.5.7. EXECUTE Command

The EXECUTE command is used to execute a procedure, such as a virtual procedure or a stored procedure. Procedures may have zero or more scalar input parameters. The return value from a procedure is a result set or the set of inout/out/return scalars. Note that EXEC or CALL can be used as a short form of this command.

Example Syntax

- EXECUTE proc()
- CALL proc(value, ...)
- EXECUTE proc(name1=>value1,name4=>param4, ...) - named parameter syntax

Syntax Rules:

- The default order of parameter specification is the same as how they are defined in the procedure definition.
- You can specify the parameters in any order by name. Parameters that have default values and/or are nullable in the metadata, can be omitted from the named parameter call and will have the appropriate value passed at runtime.
- Positional parameters that are have default values and/or are nullable in the metadata, can be omitted from the end of the parameter list and will have the appropriate value passed at runtime.
- If the procedure does not return a result set, the values from the RETURN, OUT, and IN_OUT parameters will be returned as a single row when used as an inline view query.
- A VARIADIC parameter may be repeated 0 or more times as the last positional argument.

3.5.8. Procedural Relational Command

Procedural relational commands use the syntax of a SELECT to emulate an EXEC. In a procedural relational command a procedure group name is used in a FROM clause in place of a table. That procedure will be executed in place of normal table access if all of the necessary input values can be found in criteria against the procedure. Each combination of input values found in the criteria results in an execution of the procedure.

Example Syntax

- SELECT * FROM proc
- SELECT output_param1, output_param2 FROM proc WHERE input_param1 = 'x'
- SELECT output_param1, output_param2 FROM proc, table WHERE input_param1 = table.col1 AND input_param2 = table.col2

Syntax Rules:

- The procedure as a table projects the same columns as an exec with the addition of the input parameters. For procedures that do not return a result set, IN_OUT columns will be projected as two columns, one that represents the output value and one named {column name}_IN that represents the input of the parameter.
- Input values are passed via criteria. Values can be passed by '=','is null', or 'in' predicates. Disjuncts are not allowed. It is also not possible to pass the value of a non-comparable column through an equality predicate.
- The procedure view automatically has an access pattern on its IN and IN_OUT parameters which allows it to be planned correctly as a dependent join when necessary or fail when sufficient criteria cannot be found.

- Procedures containing duplicate names between the parameters (IN, IN_OUT, OUT, RETURN) and result set columns cannot be used in a procedural relational command.
- Default values for IN, IN_OUT parameters are not used if there is no criteria present for a given input. Default values are only valid for named procedure syntax. See Section 3.5.7, “EXECUTE Command”.

NOTE

The usage of 'in' or join criteria can result in the procedure being executed multiple times.

NOTE

None of the issues listed in the syntax rules above exist if a nested table reference is used. See Section 3.6.4, “FROM Clause”.

3.5.9. Set Operations

JBoss Data Virtualization supports the UNION, UNION ALL, INTERSECT, EXCEPT set operations as ways of combining the results of query expressions.

Usage:

```
queryExpression (UNION|INTERSECT|EXCEPT) [ALL] queryExpression [ORDER
BY...]
```

Syntax Rules:

- The output columns will be named by the output columns of the first set operation branch.
- Each SELECT must have the same number of output columns and compatible data types for each relative column. Data type conversion will be performed if data types are inconsistent and implicit conversions exist.
- If UNION, INTERSECT, or EXCEPT is specified without all, then the output columns must be comparable types.
- INTERSECT ALL, and EXCEPT ALL are currently not supported.

3.5.10. Subqueries

A subquery is an SQL query embedded within another SQL query. The query containing the subquery is the outer query.

Supported subquery types:

- Scalar subquery - a subquery that returns only a single column with a single value. Scalar subqueries are a type of expression and can be used where single valued expressions are expected.
- Correlated subquery - a subquery that contains a column reference to form the outer query.
- Uncorrelated subquery - a subquery that contains no references to the outer subquery.

3.5.11. Inline Views

Subqueries in the FROM clause of the outer query (also known as "inline views") can return any number of rows and columns. This type of subquery must always be given an alias. An inline view is nearly identical to a traditional view.

```
SELECT a FROM (SELECT Y.b, Y.c FROM Y WHERE Y.d = 3) AS X WHERE a = X.c
AND b = X.b
```

See Also:

- Section 3.6.2, “WITH Clause”

3.5.12. Alternative Subquery Usage

Subqueries are supported in quantified criteria, the EXISTS predicate, the IN predicate, and as Scalar Subqueries (see Section 3.3.9, “Scalar Subqueries”).

Example 3.5. Example Subquery in WHERE Using EXISTS

```
SELECT a FROM X WHERE EXISTS (SELECT 1 FROM Y WHERE c=X.a)
```

Example 3.6. Example Quantified Comparison Subqueries

```
SELECT a FROM X WHERE a >= ANY (SELECT b FROM Y WHERE c=3)
SELECT a FROM X WHERE a < SOME (SELECT b FROM Y WHERE c=4)
SELECT a FROM X WHERE a = ALL (SELECT b FROM Y WHERE c=2)
```

Example 3.7. Example IN Subquery

```
SELECT a FROM X WHERE a IN (SELECT b FROM Y WHERE c=3)
```

See also Subquery Optimization .

3.6. DML CLAUSES

3.6.1. DML Clauses

DML clauses are used in various SQL commands (see Section 3.5.1, “DML Commands”) to specify particular relations and how to present them. Nearly all these features follow standard SQL syntax and functionality, so any SQL reference can be used for more information.

3.6.2. WITH Clause

JBoss Data Virtualization supports non-recursive common table expressions via the WITH clause. WITH clause items may be referenced as tables in subsequent WITH clause items and in the main query. The WITH clause can be thought of as providing query-scoped temporary tables.

Usage:

```
WITH name [(column, ...)] AS (query expression) ...
```

Syntax Rules:

- All of the projected column names must be unique. If they are not unique, then the column name list must be provided.
- If the columns of the WITH clause item are declared, then they must match the number of columns projected by the query expression.
- Each WITH clause item must have a unique name.

NOTE

The WITH clause is also subject to optimization and its entries may not be processed if they are not needed in the subsequent query.

3.6.3. SELECT Clause

SQL queries start with the SELECT keyword and are often referred to as "SELECT statements". JBoss Data Virtualization supports most of the standard SQL query constructs.

Usage:

```
SELECT [DISTINCT|ALL] ((expression [[AS] name])|(group
identifier.STAR))*|STAR ...
```

Syntax Rules:

- Aliased expressions are only used as the output column names and in the ORDER BY clause. They cannot be used in other clauses of the query.
- DISTINCT may only be specified if the SELECT symbols are comparable.

3.6.4. FROM Clause

The FROM clause specifies the target table(s) for SELECT, UPDATE, and DELETE statements.

Example Syntax:

- FROM table [[AS] alias]
- FROM table1 [INNER|LEFT OUTER|RIGHT OUTER|FULL OUTER] JOIN table2 ON join-criteria
- FROM table1 CROSS JOIN table2
- FROM (subquery) [AS] alias
- FROM TABLE(subquery) [AS] alias

NOTE

See Section 3.6.6, “Nested Tables”.

- FROM table1 JOIN /*+ MAKEDEP */ table2 ON join-criteria
- FROM table1 JOIN /*+ MAKENOTDEP */ table2 ON join-criteria
- FROM /*+ MAKEIND */ table1 JOIN table2 ON join-criteria
- FROM /*+ NO_UNNEST */ vw1 JOIN table2 ON join-criteria
- FROM table1 left outer join /*+ optional */ table2 ON join-criteria

NOTE

See Section 3.5.10, “Subqueries”.

- FROM TEXTTABLE...

NOTE

See Section 3.6.7, “Nested Tables: TEXTTABLE”.

- FROM XMLTABLE...

NOTE

See Section 3.6.8, “Nested Tables: XMLTABLE”.

- FROM ARRAYTABLE...

NOTE

See Section 3.6.9, “Nested Tables: ARRAYTABLE”.

- FROM OBJECTTABLE...

NOTE

See Section 3.6.10, “Nested Tables: OBJECTTABLE”.

- FROM (SELECT ...)

NOTE

See Section 3.5.10, “Subqueries”.

3.6.5. FROM Clause Hints

From clause hints are typically specified in a comment block. If multiple hints apply, they should be placed in the same comment block. For example:

```
FROM /*+ MAKEDEP PRESERVE */ (tbl1 inner join tbl2 inner join tbl3 on
tbl2.col1 = tbl3.col1 on tbl1.col1 = tbl2.col1), tbl3 WHERE tbl1.col1 =
tbl2.col1
```

Dependent Joins Hints

MAKEIND, MAKEDEP, and MAKENOTDEP are hints used to control dependent join behavior (see Section 14.7.3, "Dependent Joins"). They should only be used in situations where the optimizer does not choose the most optimal plan based upon query structure, metadata, and costing information. The hints may appear in a comment following the FROM keyword. The hints can be specified against any FROM clause, not just a named table.

NO_UNNEST

NO_UNNEST can be specified against a FROM clause or view to instruct the planner not to merge the nested SQL in the surrounding query - also known as view flattening. This hint only applies to JBoss Data Virtualization planning and is not passed to source queries. NO_UNNEST may appear in a comment following the FROM keyword.

PRESERVE

The PRESERVE hint can be used against an ANSI join tree to preserve the structure of the join rather than allowing the JBoss Data Virtualization optimizer to reorder the join. This is similar in function to the Oracle ORDERED or MySQL STRAIGHT_JOIN hints.

```
FROM /*+ PRESERVE */ (tbl1 inner join tbl2 inner join tbl3 on tbl2.col1
= tbl3.col1 on tbl1.col1 = tbl2.col1)
```

3.6.6. Nested Tables

Nested tables may appear in the FROM clause with the TABLE keyword. They are an alternative to using a view with normal join semantics. The columns projected from the command contained in the nested table may be used just as any of the other FROM clause projected columns in join criteria, the where clause, etc.

A nested table may have correlated references to preceding FROM clause column references as long as INNER and LEFT OUTER joins are used. This is especially useful in cases where the nested expression is a procedure or function call.

Valid example:

```
select * from t1, TABLE(call proc(t1.x)) t2
```

Invalid example, since t1 appears after the nested table in the FROM clause:

```
select * from TABLE(call proc(t1.x)) t2, t1
```

NOTE

The usage of a correlated nested table may result in multiple executions of the table expression - once for each correlated row.

3.6.7. Nested Tables: TEXTTABLE

The TEXTTABLE function processes character input to produce tabular output. It supports both fixed and delimited file format parsing. The function itself defines what columns it projects. The TEXTTABLE function is implicitly a nested table and may be used within FROM clauses.

```
TEXTTABLE(expression COLUMNS <COLUMN>, ... [NO ROW DELIMITER] [DELIMITER
char] [(QUOTE|ESCAPE) char] [HEADER [integer]] [SKIP integer]) AS name
```

```
COLUMN := name datatype [WIDTH integer [NO TRIM]]
```

Parameters

- expression is the text content to process, which should be convertible to CLOB.
- SELECTOR specifies that delimited lines should only match if the line begins with the selector string followed by a delimiter. The selector value is a valid column value. If a TEXTTABLE SELECTOR is specified, a SELECTOR may also be specified for column values. A column SELECTOR argument will select the nearest preceding text line with the given SELECTOR prefix and select the value at the given 1-based integer position (which includes the selector itself). If no such text line or position with a given line exists, a null value will be produced.
- NO ROW DELIMITER indicates that fixed parsing should not assume the presence of newline row delimiters.
- DELIMITER sets the field delimiter character to use. Defaults to ','.
- QUOTE sets the quote, or qualifier, character used to wrap field values. Defaults to '"'.
- ESCAPE sets the escape character to use if no quoting character is in use. This is used in situations where the delimiter or new line characters are escaped with a preceding character, e.g. \,
- HEADER specifies the text line number (counting every new line) on which the column names occur. All lines prior to the header will be skipped. If HEADER is specified, then the header line will be used to determine the TEXTTABLE column position by case-insensitive name matching. This is especially useful in situations where only a subset of the columns are needed. If the HEADER value is not specified, it defaults to 1. If HEADER is not specified, then columns are expected to match positionally with the text contents.
- SKIP specifies the number of text lines (counting every new line) to skip before parsing the contents. You can still specify a HEADER with SKIP.
- WIDTH indicates the fixed-width length of a column in characters - not bytes. The CR NL newline value counts as a single character.
- NO TRIM specifies that the text value should not be trimmed of all leading and trailing white space.

Syntax Rules:

- If width is specified for one column it must be specified for all columns and be a non-negative integer.

- If width is specified, then fixed width parsing is used and ESCAPE, QUOTE, and HEADER should not be specified.
- If width is not specified, then NO ROW DELIMITER cannot be used.
- The column names must not contain duplicates.

Examples

- Use of the HEADER parameter, returns 1 row ['b']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('col1,col2,col3\na,b,c') COLUMNS
col2 string HEADER) x
```

- Use of fixed width, returns 2 rows ['a', 'b', 'c'], ['d', 'e', 'f']:

```
SELECT * FROM TEXTTABLE(UNESCAPE('abc\ndef') COLUMNS col1 string
width 1, col2 string width 1, col3 string width 1) x
```

- Use of fixed width without a row delimiter, returns 3 rows ['a'], ['b'], ['c']:

```
SELECT * FROM TEXTTABLE('abc' COLUMNS col1 string width 1 NO ROW
DELIMITER) x
```

- Use of ESCAPE parameter, returns 1 row ['a,', 'b']:

```
SELECT * FROM TEXTTABLE('a:,,b' COLUMNS col1 string, col2 string
ESCAPE ':') x
```

- As a nested table:

```
SELECT x.* FROM t, TEXTTABLE(t.clobcolumn COLUMNS first string,
second date SKIP 1) x
```

- Use of SELECTOR, returns 2 rows ['c', 'd', 'b'], ['c', 'f', 'b']:

```
SELECT * FROM TEXTTABLE('a,b\nc,d\nc,f' SELECTOR 'c' COLUMNS col1
string, col2 string col3 string SELECTOR 'a' 2) x
```

3.6.8. Nested Tables: XMLTABLE

The XMLTABLE function uses XQuery to produce tabular output. The XMLTABLE function is implicitly a nested table and may be used within FROM clauses. XMLTABLE is part of the SQL/XML 2006 specification.

Usage:

```
XMLTABLE([<NSP>,] xquery-expression [<PASSING>] [COLUMNS <COLUMN>, ... )]
AS name
```

```
COLUMN := name (FOR ORDINALITY | (datatype [DEFAULT expression] [PATH
string]))
```

See XMLELEMENT for the definition of NSP - XMLNAMESPACES.

See XMLQUERY for the definition of PASSING.

NOTE

See also XQuery Optimization.

Parameters

- The optional XMLNAMESPACES clause specifies the namespaces for use in the XQuery and COLUMN path expressions.
- The xquery-expression must be a valid XQuery. Each sequence item returned by the xquery will be used to create a row of values as defined by the COLUMNS clause.
- If COLUMNS is not specified, then that is the same as having the COLUMNS clause: "COLUMNS OBJECT_VALUE XML PATH '.'", which returns the entire item as an XML value.
- A FOR ORDINALITY column is typed as integer and will return the one-based item number as its value.
- Each non-ordinality column specifies a type and optionally a PATH and a DEFAULT expression.
- If PATH is not specified, then the path will be the same as the column name.

Syntax Rules:

- Only 1 FOR ORDINALITY column may be specified.
- The columns names must not contain duplicates.
- The blob data type is supported, but there is only built-in support for xs:hexBinary values. For xs:base64Binary, use a workaround of a PATH that uses the explicit value constructor "xs:base64Binary(<path>)".

Examples

- Use of passing, returns 1 row [1]:

```
select * from xmltable('/a' PASSING xmlparse(document '<a id="1"/>')
COLUMNS id integer PATH '@id') x
```

- As a nested table:

```
select x.* from t, xmltable('/x/y' PASSING t.doc COLUMNS first
string, second FOR ORDINALITY) x
```

3.6.9. Nested Tables: ARRAYTABLE

The ARRAYTABLE function processes an array input to produce tabular output. The function itself defines what columns it projects. The ARRAYTABLE function is implicitly a nested table and may be used within FROM clauses.

Usage:

```
ARRAYTABLE(expression COLUMNS <COLUMN>, ...) AS name
```

```
COLUMN := name datatype
```

Parameters

- expression - the array to process, which should be a java.sql.Array or java array value.

Syntax Rules:

- The columns names must not contain duplicates.

Examples

- As a nested table:

  ```
  select x.* from (call source.invokeMDX('some query')) r,
  arraytable(r.tuple COLUMNS first string, second bigdecimal) x
  ```

ARRAYTABLE is effectively a shortcut for using the array_get function (see Section 3.4.18, “Miscellaneous Functions”) in a nested table. For example:

```
ARRAYTABLE(val COLUMNS col1 string, col2 integer) AS X
```

is the same as

```
TABLE(SELECT cast(array_get(val, 1) AS string) AS col1,
cast(array_get(val, 2) AS integer) AS col2) AS X
```

3.6.10. Nested Tables: OBJECTTABLE

The OBJECTTABLE function processes an object input to produce tabular output. The function itself defines what columns it projects. The OBJECTTABLE function is implicitly a nested table and may be correlated to preceding FROM clause entries.

Usage:

```
OBJECTTABLE([LANGUAGE lang] rowScript [PASSING val AS name ...] COLUMNS
colName colType colScript [DEFAULT defaultExpr] ...) AS id
```

Parameters

- lang - an optional string literal that is the case sensitive language name of the scripts to be processed. The script engine must be available via a JSR-223 ScriptEngineManager lookup. In some instances this may mean making additional modules available to your VDB, which can be done via the same process as adding modules/libraries for UDFs (see Non-Pushdown Support for User-Defined Functions in the *Development Guide: Server Development*). If a LANGUAGE is not specified, the default of 'teiid_script' (see below) will be used.

- name - an identifier that will bind the val expression value into the script context.

- rowScript is a string literal specifying the script to create the row values. For each non-null item the Iterator produces the columns will be evaluated.
- colName/colType are the id/data type of the column, which can optionally be defaulted with the DEFAULT clause expression defaultExpr.
- colScript is a string literal specifying the script that evaluates to the column value.

Syntax Rules:

- The column names must be not contain duplicates.
- JBoss Data Virtualization will place several special variables in the script execution context. The CommandContext is available as teiid_context. Additionally the colScripts may access teiid_row and teiid_row_number. teiid_row is the current row object produced by the row script. teiid_row_number is the current 1-based row number.
- rowScript is evaluated to an Iterator. If the results is already an Iterator, it is used directly. If the evaluation result is an Iteratable, then an Iterator will be obtained. Any other Object will be treated as an Iterator of a single item). In all cases null row values will be skipped.

NOTE

While there is no restriction what can be used as a PASSING variable names you should choose names that can be referenced as identifiers in the target language.

Examples

- Accessing special variables:

```
SELECT x.* FROM OBJECTTABLE('teiid_context' COLUMNS "user" string
'teiid_row.userName', row_number integer 'teiid_row_number') AS x
```

The result would be a row with two columns containing the user name and 1 respectively.

NOTE

Due to their mostly unrestricted access to Java functionality, usage of languages other than teiid_script is restricted by default. A VDB must declare all allowable languages by name in the allowed-languages VDB property (see Section 9.1, “VDB Definition”) using a comma separated list. The names are case sensitive names and should be separated without whitespace. Without this property it is not possible to use OBJECTTABLE even from within view definitions that are not subject to normal permission checks. Data Roles are also secured with User Query Permissions.

teiid_script

teiid_script is a simple scripting expression language that allows access to passing and special variables as well as any non-void 0-argument methods on objects. A teiid_script expression begins by referencing the passing or special variable. Then any number of .method accessors may be chained to evaluate the expression to a different value. Methods may be accessed by their property names, for example foo rather than getFoo. If the object both a `getFoo()` and `foo()` method, then the accessor foo references `foo()` and getFoo should be used to call the getter.

teiid_script is effectively dynamically typed as typing is performed at runtime. If a accessor does not exist on the object or if the method is not accessible, then an exception will be raised.

Examples

- To get the VDB description string:

  ```
  teiid_context.session.vdb.description
  ```

3.6.11. WHERE Clause

The WHERE clause defines the criteria to limit the records affected by SELECT, UPDATE, and DELETE statements.

Usage:

```
WHERE criteria
```

See Also:

- Section 3.3.11, “Criteria”

3.6.12. GROUP BY Clause

The GROUP BY clause denotes that rows should be grouped according to the specified expression values. One row will be returned for each group, after optionally filtering those aggregate rows based on a HAVING clause.

Usage:

```
GROUP BY expression (,expression)*
```

Syntax Rules:

- Column references in the GROUP BY clause must be unaliased output columns.
- Expressions used in the GROUP BY clause must appear in the SELECT clause.
- Column references and expressions in the SELECT clause that are not used in the GROUP BY clause must appear in aggregate functions.
- If an aggregate function is used in the SELECT clause and no GROUP BY is specified, an implicit GROUP BY will be performed with the entire result set as a single group. In this case, every column in the SELECT must be an aggregate function as no other column value will be fixed across the entire group.
- The group by columns must be of a comparable type.

3.6.13. HAVING Clause

The HAVING clause operates exactly as a WHERE clause although it operates on the output of a GROUP BY. It supports the same syntax as the WHERE clause.

Syntax Rules:

- Expressions used in the GROUP BY clause must either contain an aggregate function: COUNT, AVG, SUM, MIN, MAX. or be one of the grouping expressions.

3.6.14. ORDER BY Clause

The ORDER BY clause specifies how records should be sorted. The options are ASC (ascending) and DESC (descending).

Usage:

```
ORDER BY expression [ASC|DESC] [NULLS (FIRST|LAST)], ...
```

Syntax Rules:

- Sort columns may be specified positionally by a 1-based positional integer, by SELECT clause alias name, by SELECT clause expression, or by an unrelated expression.
- Column references may appear in the SELECT clause as the expression for an aliased column or may reference columns from tables in the FROM clause. If the column reference is not in the SELECT clause the query must not be a set operation, specify SELECT DISTINCT, or contain a GROUP BY clause.
- Unrelated expressions, expressions not appearing as an aliased expression in the SELECT clause, are allowed in the ORDER BY clause of a non-set QUERY. The columns referenced in the expression must come from the FROM clause table references. The column references cannot be to alias names or positional.
- The ORDER BY columns must be of a comparable type.
- If an ORDER BY is used in an inline view or view definition without a LIMIT clause, it will be removed by the JBoss Data Virtualization optimizer.
- If NULLS FIRST/LAST is specified, then nulls are guaranteed to be sorted either first or last. If the null ordering is not specified, then results will typically be sorted with nulls as low values, which is the JBoss Data Virtualization internal default sorting behavior. However not all sources return results with nulls sorted as low values by default, and JBoss Data Virtualization may return results with different null orderings.

WARNING

The use of positional ordering is no longer supported by the ANSI SQL standard and is a deprecated feature in JBoss Data Virtualization. It is preferable to use alias names in the ORDER BY clause.

3.6.15. LIMIT Clause

The LIMIT clause specifies a limit on the number of records returned from the SELECT command. An optional offset (the number of rows to skip) can be specified. The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses. If an ORDER BY is also specified, it will be applied before the OFFSET/LIMIT are applied. If an ORDER BY is not specified there is generally no guarantee what subset of rows will be returned.

Usage:

```
LIMIT [offset,] limit
```

```
[OFFSET offset ROW|ROWS] [FETCH FIRST|NEXT [limit] ROW|ROWS ONLY
```

Syntax Rules:

- The limit/offset expressions must be a non-negative integer or a parameter reference (?). An offset of 0 is ignored. A limit of 0 will return no rows.
- The terms FIRST/NEXT are interchangeable as well as ROW/ROWS.
- The LIMIT clause may take an optional preceding NON_STRICT hint to indicate that push operations should not be inhibited even if the results will not be consistent with the logical application of the limit. The hint is only needed on unordered limits, e.g. "SELECT * FROM VW /*+ NON_STRICT */ LIMIT 2".

Examples:

- LIMIT 100 - returns the first 100 records (rows 1-100)
- LIMIT 500, 100 - skips 500 records and returns the next 100 records (rows 501-600)
- OFFSET 500 ROWS - skips 500 records
- OFFSET 500 ROWS FETCH NEXT 100 ROWS ONLY - skips 500 records and returns the next 100 records (rows 501-600)
- FETCH FIRST ROW ONLY - returns only the first record

3.6.16. INTO Clause

WARNING

Usage of the INTO Clause for inserting into a table has been been deprecated. An INSERT with a query command should be used instead. Refer to Section 3.5.3, "INSERT Command".

3.6.17. OPTION Clause

The OPTION keyword denotes options the user can pass in with the command. These options are specific to JBoss Data Virtualization and not covered by any SQL specification.

Usage:

```
OPTION option, (option)*
```

Supported options:

- MAKEDEP table [(,table)*] - specifies source tables that will be made dependent in the join
- MAKENOTDEP table [(,table)*] - prevents a dependent join from being used
- NOCACHE [table (,table)*] - prevents cache from being used for all tables or for the given tables

Examples:

- OPTION MAKEDEP table1
- OPTION NOCACHE

All tables specified in the OPTION clause should be fully qualified, however the name may match either an alias name or the fully qualified name.

NOTE

Previous versions of JBoss Data Virtualization accepted the PLANONLY, DEBUG, and SHOWPLAN option arguments. These are no longer accepted in the OPTION clause. See *Red Hat JBoss Data Virtualization Development Guide: Client Development* for replacements to those options.

3.7. DDL COMMANDS

3.7.1. DDL Commands

JBoss Data Virtualization supports a subset of DDL to create/drop temporary tables and to manipulate procedure and view definitions at runtime. It is not currently possible to arbitrarily drop/create non-temporary metadata entries. See Section 12.1, “DDL Metadata” for DDL usage to define schemas within a VDB.

NOTE

A **`MetadataRepository`** must be configured to make a non-temporary metadata update persistent. See Runtime Metadata Updates in Red Hat JBoss Data Virtualization *Development Guide: Server Development* for more information.

3.7.2. Temporary Tables

JBoss Data Virtualization supports creating temporary tables. Temporary tables are dynamically created, but are treated as any other physical table.

Temporary tables can be defined implicitly by referencing them in a INSERT statement or explicitly with a CREATE TABLE statement. Implicitly created temporary tables must have a name that starts with '#'.

Creation syntax:

- Temporary tables can be defined explicitly with a CREATE TABLE statement:

  ```
  CREATE LOCAL TEMPORARY TABLE name (column type [NOT NULL], ...
  [PRIMARY KEY (column, ...)])
  ```

  Alternatively, temporary tables can be defined implicitly by referencing them in a INSERT statement. Implicitly created temporary tables must have a name that starts with '#':

  ```
  INSERT INTO #name (column, ...) VALUES (value, ...)
  ```

  NOTE

  If #name does not exist, it will be defined using the given column names and types from the value expressions.

  ```
  INSERT INTO #name [(column, ...)] select c1, c2 from t
  ```

  NOTE

  If #name does not exist, it will be defined using the target column names and the types from the query derived columns. If target columns are not supplied, the column names will match the derived column names from the query.

  Use the SERIAL data type to specify a NOT NULL and auto-incrementing INTEGER column. The starting value of a SERIAL column is 1.

Drop syntax:

- DROP TABLE name

Primary Key Support

- All key columns must be comparable.
- Use of a primary key creates a clustered index that supports search improvements for comparison, in, like, and order by.
- Null is an allowable primary key value, but there must be only 1 row that has an all null key.

```
Limitations:
```

- With the CREATE TABLE syntax only basic table definition (column name and type information) and an optional primary key are supported.
- The "ON COMMIT" clause is not supported in the CREATE TABLE statement.
- "drop behavior" option is not supported in the drop statement.
- Only local temporary tables are supported. This implies that the scope of temp table will be either to the session or the block of a virtual procedure that creates it.
- Session level temporary tables are not fail-over safe.

- Temp tables support a READ_UNCOMMITED transaction isolation level. There are no locking mechanisms available to support higher isolation levels and the result of a rollback may be inconsistent across multiple transactions. If concurrent transactions are not associated with the same local temporary table or session, then the transaction isolation level is effectively SERIALIZABLE. If you want full consistency with local temporary tables, then only use a connection with 1 transaction at a time. This mode of operation is ensured by connection pooling that tracks connections by transaction.

- LOB values (XML, CLOB, BLOB) are tracked by reference rather than by value in a temporary table. LOB values from external sources that are inserted in a temporary table may become unreadable when the associated statement or connection is closed.

The following example is a series of statements that loads a temporary table with data from 2 sources, and with a manually inserted record, and then uses that temp table in a subsequent query.

```
...
CREATE LOCAL TEMPORARY TABLE TEMP (a integer, b integer, c integer);
SELECT * INTO temp FROM Src1; SELECT * INTO temp FROM Src2;
INSERT INTO temp VALUES (1,2,3);
SELECT a,b,c FROM Src3, temp WHERE Src3.a = temp.b;
...
```

See Section 3.10.1, “Virtual Procedures” for more on temporary table usage.

3.7.3. Foreign Temporary Tables

Unlike a local temporary table, a foreign temporary table is a reference to an actual source table that is created at runtime rather than during the metadata load.

A foreign temporary table requires explicit creation syntax:

```
CREATE FOREIGN TEMPORARY TABLE name ... ON schema
```

Where the table creation body syntax is the same as a standard CREATE FOREIGN TABLE DDL statement (see Section 12.1, “DDL Metadata”). In general usage of DDL OPTION, clauses may be required to properly access the source table, including setting the name in source, updatability, native types, etc.

The schema name must specify an existing schema/model in the VDB. The table will be accessed as if it is on that source, however within JBoss Data Virtualization the temporary table will still be scoped the same as a non-foreign temporary table. This means that the foreign temporary table will not belong to a JBoss Data Virtualization schema and will be scoped to the session or procedure block where created.

The DROP syntax for a foreign temporary table is the same as for a non-foreign temporary table.

Neither a CREATE nor a corresponding DROP of a foreign temporary table issue a pushdown command, rather this mechanism simply exposes a source table for use within JBoss Data Virtualization on a temporary basis.

There are two usage scenarios for a FOREIGN TEMPORARY TABLE. The first is to dynamically access additional tables on the source. The other is to replace the usage of a JBoss Data Virtualization local temporary table for performance reasons. The usage pattern for the latter case would look like:

```
//- create the source table
```

```
call source.native("CREATE GLOBAL TEMPORARY TABLE name IF NOT EXISTS ON
COMMIT DELETE ROWS");
//- bring the table into JBoss Data Virtualization
CREATE FOREIGN TEMPORARY TABLE name ... OPTIONS (UPDATABLE true)
//- use the table
...
//- forget the table
DROP TABLE name
```

Note the usage of the native procedure to pass source specific CREATE ddl to the source. JBoss Data Virtualization does not currently attempt to pushdown a source creation of a temporary table based upon the CREATE statement. Some other mechanism, such as the native procedure shown above, must be used to first create the table. Also note the table is explicitly marked as updatable, since DDL defined tables are not updatable by default.

The source's handling of temporary tables must also be understood to make this work as intended. Sources that use the same GLOBAL table definition for all sessions while scoping the data to be session specific (such as Oracle) or sources that support session scoped temporary tables (such as PostgreSQL) will work if accessed under a transaction. A transaction is necessary because:

- the source on commit behavior (most likely DELETE ROWS or DROP) will ensure clean-up. Keep in mind that a JBoss Data Virtualization DROP does not issue a source command and is not guaranteed to occur (in some exception cases, loss of DB connectivity, hard shutdown, etc.).
- the source pool when using track connections by transaction will ensure that multiple uses of that source by JBoss Data Virtualization will use the same connection/session and thus the same temporary table and data.

NOTE

Since the ON COMMIT clause is not yet supported by JBoss Data Virtualization, it is important to consider that the source table ON COMMIT behavior will likely be different that the default, PRESERVE ROWS, for JBoss Data Virtualization local temporary tables.

3.7.4. Alter View

Usage:

```
ALTER VIEW name AS queryExpression
```

Syntax Rules:

- The alter query expression may be prefixed with a cache hint for materialized view definitions. The hint will take effect the next time the materialized view table is loaded.

3.7.5. Alter Procedure

Usage:

```
ALTER PROCEDURE name AS block
```

Syntax Rules:

- The alter block should not include 'CREATE VIRTUAL PROCEDURE'
- The alter block may be prefixed with a cache hint for cached procedures.

3.7.6. Create Trigger

Usage:

```
CREATE TRIGGER ON name INSTEAD OF INSERT|UPDATE|DELETE AS FOR EACH ROW
block
```

Syntax Rules:

- The target, name, must be an updatable view.
- An INSTEAD OF TRIGGER must not yet exist for the given event.
- Triggers are not yet true schema objects. They are scoped only to their view and have no name.

Limitations:

- There is no corresponding DROP operation. See Section 3.7.7, “Alter Trigger” for enabling/disabling an existing trigger.

3.7.7. Alter Trigger

Usage:

```
ALTER TRIGGER ON name INSTEAD OF INSERT|UPDATE|DELETE (AS FOR EACH ROW
block) | (ENABLED|DISABLED)
```

Syntax Rules:

- The target, name, must be an updatable view.
- Triggers are not yet true schema objects. They are scoped only to their view and have no name.
- Update Procedures must already exist for the given trigger event. See Section 3.10.6, “Update Procedures”.

NOTE

If the default inherent update is chosen in Teiid Designer, any SQL associated with update (shown in a greyed out text box) is not part of the VDB and cannot be enabled with an alter trigger statement.

3.8. XML DOCUMENT GENERATION

3.8.1. XML Document Generation

XML documents can be constructed dynamically using XML Document Models. A document model is generally created from a schema. The document model is bound to relevant SQL statements through mapping classes. See the *Red Hat JBoss Data Virtualization User Guide*for more information about creating document models.

Querying XML documents is similar to querying relational tables. An idiomatic SQL variant with special scalar functions provides control over which parts of a given document to return.

NOTE

XML documents may also be created via XQuery with the XMLQuery function or with various other SQL/XML functions. See Section 3.4.14, "XML Functions".

3.8.2. XML SELECT Command

A valid XML SELECT Command against a document model is of the form:

```
SELECT ... FROM ... [WHERE ...] [ORDER BY ...]
```

The use of any other SELECT clause is not allowed.

The fully qualified name for an XML element is:

```
"model"."document name".[path to element]."element name"
```

The fully qualified name for an attribute is:

```
"model"."document name".[path to element]."element name".[@]"attribute
name"
```

Partially qualified names for elements and attributes can be used as long as the partial name is unique.

3.8.3. XML SELECT: FROM Clause

This clause specifies the document to generate. Document names resemble other virtual groups - **`"model"."document name"`**.

Syntax Rules:

- The FROM clause must contain only one unary clause specifying the desired document.

3.8.4. XML SELECT: SELECT Clause

The SELECT clause determines which parts of the XML document are generated for output.

Example Syntax:

- select * from model.doc
- select model.doc.root.parent.element.* from model.doc
- select element, element1.@attribute from model.doc

Syntax Rules:

- SELECT * and SELECT "xml" are equivalent and specify that every element and attribute of the document should be output.
- The SELECT clause of an XML Query may only contain *, "xml", or element and attribute references from the specified document. Any other expressions are not allowed.
- If the SELECT clause contains an element or attribute reference (other than * or "xml") then only the specified elements, attributes, and their ancestor elements will be in the generated document.
- element.* specifies that the element, its attribute, and all child content should be output.

3.8.5. XML SELECT: WHERE Clause

The WHERE clause specifies how to filter content from the generated document based upon values contained in the underlying mapping classes. Most predicates are valid in an XML SELECT Command, however combining value references from different parts of the document may not always be allowed.

Criteria is logically applied to a context which is directly related to a mapping class. Starting with the root mapping class, there is a root context that describes all of the top level repeated elements that will be in the output document. Criteria applied to the root or any other context will change the related mapping class query to apply the affects of the criteria, which can include checking values from any of the descendant mapping classes.

Example Syntax:

- select element, element1.@attribute from model.doc where element1.@attribute = 1
- select element, element1.@attribute from model.doc where context(element1, element1.@attribute) = 1

Syntax Rules:

- Each criteria conjunct must refer to a single context and can be criteria that applies to a mapping class, contain a `rowlimit` function, or contain `rowlimitexception` function. Refer to Section 3.8.9, “ROWLIMIT Function” and Section 3.8.10, “ROWLIMITEXCEPTION Function”.
- Criteria that applies to a mapping class is associated to that mapping class using the `context` function. The absence of a context function implies the criteria applies to the root context. Refer to Section 3.8.8, “CONTEXT Function”.
- At a given context the criteria can span multiple mapping classes provided that all mapping classes involved are either parents of the context, the context itself, or a descendant of the context.

NOTE

Implied root context user criteria against a document model with sibling root mapping classes is not generally semantically correct. It is applied as if each of the conjuncts is applied to only a single root mapping class. This behavior is the same as prior releases but may be fixed in a future release.

3.8.6. XML SELECT: ORDER BY Clause

The XML SELECT Command ORDER BY clause specifies ordering for the referenced mapping class queries.

Syntax Rules:

- Each ORDER BY item must be an element or attribute reference tied a output value from a mapping class.
- The order of the ORDER BY items is the relative order applied to their respective mapping classes.

3.8.7. XML SELECT Command Specific Functions

XML SELECT Command functions resemble scalar functions, but act as hints in the WHERE clause:

- CONTEXT Function
- ROWLIMIT Function
- ROWLIMITEXCEPTION Function

These functions are only valid in an XML SELECT Command.

3.8.8. CONTEXT Function

This function selects the context for the containing conjunct.

```
CONTEXT(arg1, arg2)
```

Syntax Rules:

- Context functions apply to the whole conjunct.
- The first argument must be an element or attribute reference from the mapping class whose context the criteria conjunct will apply to.
- The second parameter is the return value for the function.

3.8.9. ROWLIMIT Function

This function limits the rows processed for the given context.

```
ROWLIMIT(arg)
```

Syntax Rules:

- The first argument must be an element or attribute reference from the mapping class whose context the row limit applies.
- The ROWLIMIT function must be used in equality comparison criteria with the right hand expression equal to an positive integer number or rows to limit.
- Only one row limit or row limit exception may apply to a given context.

3.8.10. ROWLIMITEXCEPTION Function

This function limits the rows processed for the given context and throws an exception if the given number of rows is exceeded.

```
ROWLIMITEXCEPTION(arg)
```

Syntax Rules:

- The first argument must be an element or attribute reference from the mapping class whose context the row limit exception applies.
- The ROWLIMITEXCEPTION function must be used in equality comparison criteria with the right hand expression equal to an positive integer number or rows to limit.
- Only one row limit or row limit exception may apply to a given context.

3.8.11. Document Generation

Document generation starts with the root mapping class and proceeds iteratively and hierarchically over all of the child mapping classes. This can result in a large number of query executions. For example if a document has a root mapping class with 3 child mapping classes. Then for each row selected by the root mapping class after the application of the root context criteria, each of the child mapping classes queries will also be executed.

NOTE

By default, XML generated by XML documents are not checked for correctness vs. the relevant schema. It is possible that the mapping class queries, the usage of specific SELECT or WHERE clause values will generate a document that is not valid with respect to the schema. Refer to Section 3.8.12, “Document Validation” for information to ensure correctness.

Sibling or cousin elements defined by the same mapping class that do not have a common parent in that mapping class will be treated as independent mapping classes during planning and execution. This allows for a more document-centric approach when applying WHERE criteria and ORDER BY clauses to mapping classes.

3.8.12. Document Validation

If the execution property `XMLValidation` is set to 'true' generated documents will be checked for correctness. However, correctness checking will not prevent invalid documents from being generated, since correctness is checked after generation.

3.9. PROCEDURAL LANGUAGE

3.9.1. Procedural Language

JBoss Data Virtualization supports a procedural language for defining virtual procedures. These are similar to stored procedures in relational database management systems. You can use this language to define the transformation logic for decomposing INSERT, UPDATE, and DELETE commands against views; these are known as update procedures. See Section 3.10.1, “Virtual Procedures” and Section 3.10.6, “Update Procedures” for more information.

3.9.2. Command Statement

A command statement executes a DML command, DDL command or dynamic SQL against one or more data sources. See Section 3.5.1, “DML Commands” and Section 3.7.1, “DDL Commands”.

Usage:

```
command [(WITH|WITHOUT) RETURN];
```

Example 3.8. Example Command Statements

```
SELECT * FROM MySchema.MyTable WHERE ColA > 100 WITHOUT RETURN;
INSERT INTO MySchema.MyTable (ColA,ColB) VALUES (50, 'hi');
```

EXECUTE commands may access IN/OUT, OUT, and RETURN parameters. To access the return value the statement will have the form **`var = EXEC proc...`**. To access OUT or IN/OUT values named parameter syntax must be used. For example, **`EXEC proc(in_param=>'1', out_param=>var)`** will assign the value of the out parameter to the variable var. It is expected that the data type of parameter will be implicitly convertible to the data type of the variable.

The RETURN clause determines if the result of the command is returnable from the procedure. WITH RETURN is the default. If the command does not return a result set or the procedure does not return a result set, the RETURN clause is ignored. If WITH RETURN is specified, the result set of the command must match the expected result set of the procedure. Only the last successfully executed statement executed WITH RETURN will be returned as the procedure result set. If there are no returnable result sets and the procedure declares that a result set will be returned, then an empty result set is returned.

3.9.3. Dynamic SQL

Dynamic SQL allows for the execution of an arbitrary SQL command in a virtual procedure. Dynamic SQL is useful in situations where the exact command form is not known prior to execution.

Usage:

```
EXECUTE IMMEDIATE <expression> [AS <variable> <type> [, <variable>
<type>]* [INTO <variable>]] [USING <variable>=<expression> [,<variable>=
<expression>]*] [UPDATE <literal>]
```

Syntax Rules:

- The "AS" clause is used to define the projected symbols names and types returned by the executed SQL string. The "AS" clause symbols will be matched positionally with the symbols returned by the executed SQL string. Non-convertible types or too few columns returned by the executed SQL string will result in an error.

- The "INTO" clause will project the dynamic SQL into the specified temp table. With the "INTO" clause specified, the dynamic command will actually execute a statement that behaves like an INSERT with a QUERY EXPRESSION. If the dynamic SQL command creates a temporary table with the "INTO" clause, then the "AS" clause is required to define the table's metadata.

- The "USING" clause allows the dynamic SQL string to contain variable references that are bound at runtime to specified values. This allows for some independence of the SQL string from the surrounding procedure variable names and input names. In the dynamic command "USING" clause, each variable is specified by short name only. However in the dynamic SQL the "USING" variable must be fully qualified to "DVAR.". The "USING" clause is only for values that will be used in the dynamic SQL as legal expressions. It is not possible to use the "USING" clause to replace table names, keywords, etc. This makes using symbols equivalent in power to normal bind (?) expressions in prepared statements. The "USING" clause helps reduce the amount of string manipulation needed. If a reference is made to a USING symbol in the SQL string that is not bound to a value in the "USING" clause, an exception will occur.

- The "UPDATE" clause is used to specify the updating model count. Accepted values are (0,1,*). 0 is the default value if the clause is not specified. See Section 6.3, "Updating Model Count".

Example 3.9. Example Dynamic SQL

```
...
/* Typically complex criteria would be formed based upon inputs to the
procedure.
 In this simple example the criteria is references the using clause to
isolate
 the SQL string from referencing a value from the procedure directly */
DECLARE string criteria = 'Customer.Accounts.Last = DVARS.LastName';

/* Now we create the desired SQL string */
DECLARE string sql_string = 'SELECT ID, First || '' '' || Last AS Name,
Birthdate FROM Customer.Accounts WHERE ' || criteria;

/* The execution of the SQL string will create the #temp table with the
columns (ID, Name, Birthdate).
  Note that we also have the USING clause to bind a value to LastName,
which is referenced in the criteria. */
EXECUTE IMMEDIATE sql_string AS ID integer, Name string, Birthdate date
INTO #temp USING LastName='some name';
/* The temp table can now be used with the values from the Dynamic SQL
*/
loop on (SELCT ID from #temp) as myCursor
...
```

Here is an example showing a more complex approach to building criteria for the dynamic SQL string. In short, the virtual procedure AccountAccess.GetAccounts has inputs ID, LastName, and bday. If a value is specified for ID it will be the only value used in the dynamic SQL criteria. Otherwise if a value is specified for LastName the procedure will detect if the value is a search string. If bday is specified in addition to LastName, it will be used to form compound criteria with LastName.

Example 3.10. Example Dynamic SQL with USING clause and dynamically built criteria string

```
...
DECLARE string crit = null;
IF (AccountAccess.GetAccounts.ID IS NOT NULL)
 crit = '(Customer.Accounts.ID = DVARS.ID)';
ELSE IF (AccountAccess.GetAccounts.LastName IS NOT NULL)
BEGIN
 IF (AccountAccess.GetAccounts.LastName == '%')
   ERROR "Last name cannot be %";
 ELSE IF (LOCATE('%', AccountAccess.GetAccounts.LastName) < 0)
   crit = '(Customer.Accounts.Last = DVARS.LastName)';
 ELSE
   crit = '(Customer.Accounts.Last LIKE DVARS.LastName)';
 IF (AccountAccess.GetAccounts.bday IS NOT NULL)
   crit = '(' || crit || ' and (Customer.Accounts.Birthdate =
DVARS.BirthDay))';
END
ELSE
 ERROR "ID or LastName must be specified.";
EXECUTE IMMEDIATE 'SELECT ID, First || '' '' || Last AS Name, Birthdate
FROM Customer.Accounts WHERE ' || crit USING
ID=AccountAccess.GetAccounts.ID,
LastName=AccountAccess.GetAccounts.LastName,
BirthDay=AccountAccess.GetAccounts.Bday;
...
```

3.9.4. Dynamic SQL Limitations

- The use of dynamic SQL command results in an assignment statement requires the use of a temp table.

Example 3.11. Example Assignment

```
EXECUTE IMMEDIATE <expression> AS x string INTO #temp;
DECLARE string VARIABLES.RESULT = (SELECT x FROM #temp);
```

- The construction of appropriate criteria will be cumbersome if parts of the criteria are not present. For example if "criteria" were already NULL, then the following example results in "criteria" remaining NULL.

Example 3.12. Example Dangerous NULL handling

```
...
criteria = '(' || criteria || ' and (Customer.Accounts.Birthdate =
DVARS.BirthDay))';
```

The preferred approach is for the user to ensure the criteria is not NULL prior its usage. If this is not possible, a good approach is to specify a default as shown in the following example.

Example 3.13. Example NULL handling

```
...
criteria = '(' || nvl(criteria, '(1 = 1)') || ' and
(Customer.Accounts.Birthdate = DVARS.BirthDay))';
```

- If the dynamic SQL is an UPDATE, DELETE, or INSERT command, and the user needs to specify the "AS" clause (which would be the case if the number of rows effected needs to be retrieved). The user will still need to provide a name and type for the return column if the into clause is specified.

Example 3.14. Example with AS and INTO clauses

```
/* This name does not need to match the expected update command
symbol "count". */
EXECUTE IMMEDIATE <expression> AS x integer INTO #temp;
```

- Unless used in other parts of the procedure, tables in the dynamic command will not be seen as sources in Teiid Designer.
- When using the "AS" clause only the type information will be available to Teiid Designer. Result set columns generated from the "AS" clause then will have a default set of properties for length, precision, etc.

3.9.5. Declaration Statement

A declaration statement declares a variable and its type. After you declare a variable, you can use it in that block within the procedure and any sub-blocks. A variable is initialized to null by default, but can also be assigned the value of an expression as part of the declaration statement.

Usage:

```
DECLARE <type> [VARIABLES.]<name> [= <expression>];
```

Example Syntax

-
  ```
  declare integer x;
  declare string VARIABLES.myvar = 'value';
  ```

Syntax Rules:

- You cannot redeclare a variable with a duplicate name in a sub-block
- The VARIABLES group is always implied even if it is not specified.
- The assignment value follows the same rules as for an Assignment Statement.
- In addition to the standard types, you may specify EXCEPTION if declaring an exception variable.

3.9.6. Assignment Statement

An assignment statement assigns a value to a variable by evaluating an expression.

Usage:

```
<variable reference> = <expression>;
```

Example Syntax

-
  ```
  myString = 'Thank you';
  VARIABLES.x = (SELECT Column1 FROM MySchema.MyTable);
  ```

Special Variables

- The **VARIABLES.ROWCOUNT** integer variable will contain the numbers of rows affected by the last INSERT/UPDATE/DELETE command statement executed. Inserts that are processed by dynamic SQL with an INTO clause will also update the ROWCOUNT.

Example 3.15. Sample Usage

```
...
UPDATE FOO SET X = 1 WHERE Y = 2;
DECLARE INTEGER UPDATED = VARIABLES.ROWCOUNT;
...
```

3.9.7. Compound Statement

A compound statement (or block) logically groups a series of statements. Temporary tables and variables created in a compound statement are local only to that block and are destroyed when exiting the block.

Usage:

```
[label :] BEGIN [[NOT] ATOMIC]
    statement*
[EXCEPTION ex
    statement*
]
END
```

NOTE

Where a block is expected by an IF, LOOP, WHILE, etc., a single statement is also accepted by the parser. Even though the block BEGIN/END are not expected, the statement will execute as if wrapped in a BEGIN/END pair.

Syntax Rules

- If NOT ATOMIC or no ATOMIC clause is specified, the block will be executed non-atomically.
- If the ATOMIC clause is specified, the block must execute atomically. If a transaction is already associated with the thread, no additional action will be taken - savepoints and/or sub-transactions are not currently used. Otherwise a transaction will be associated with the execution of the block.
- The label must not be the same as any other label used in statements containing this one.

3.9.8. Exception Handling

If the EXCEPTION clause is used within a compound statement, any processing exception emitted from statements will be caught with the flow of execution transferring to EXCEPTION statements. Any block level transaction started by this block will commit if the exception handler successfully completes. If another exception or the original exception is emitted from the exception handler the transaction will rollback. Any temporary tables or variables specific to the BLOCK will not be available to the exception handler statements.

NOTE

Only processing exceptions, which are typically caused by errors originating at the sources or with function execution, are caught. A low-level internal error or Java **`RuntimeException`** will not be caught.

To aid in the processing of a caught exception the EXCEPTION clause specifies a group name that exposes the significant fields of the exception. The exception group will contain:

Variable	Type	Description
STATE	string	The SQL State
ERRORCODE	integer	The error or vendor code. In the case of an internal exception, this will be the integer suffix of the TEIIDxxxx code
TEIIDCODE	string	The full event code. Typically TEIIDxxxx.
EXCEPTION	object	The exception being caught, will be an instance of **`TeiidSQLException`**
CHAIN	object	The chained exception or cause of the current exception

NOTE

JBoss Data Virtualization does not yet fully comply with the ANSI SQL specification on SQL State usage. For errors without an underlying SQLException cause, it is best to use the event code.

The exception group name may not be the same as any higher level exception group or loop cursor name.

Example 3.16. Example Exception Group Handling

```
BEGIN
    DECLARE EXCEPTION e = SQLEXCEPTION 'this is bad' SQLSTATE 'xxxxx';
    RAISE variables.e;
EXCEPTION e
    IF (e.state = 'xxxxx')
        //in this trivial example, we'll always hit this branch and log
the exception
        RAISE SQLWARNING e.exception;
    ELSE
        RAISE e.exception;
END
```

3.9.9. If Statement

An IF statement evaluates a condition and executes one of two statements depending on the result. You can nest IF statements to create complex branching logic. A dependent ELSE statement will execute its statement only if the IF statement evaluates to false.

Usage:

```
IF (criteria)
  block
[ELSE
  block]
END
```

Example 3.17. Example If Statement

```
IF ( var1 = 'North America')
BEGIN
  ...statement...
END ELSE
BEGIN
  ...statement...
END
```

NOTE

NULL values should be considered in the criteria of an IF statement. IS NULL criteria can be used to detect the presence of a NULL value.

3.9.10. Loop Statement

A LOOP statement is an iterative control construct that is used to cursor through a result set.

Usage:

```
[label :] LOOP ON <select statement> AS <cursorname>
  block
```

Syntax Rules

- The label must not be the same as any other label used in statements containing this one.

3.9.11. While Statement

A WHILE statement is an iterative control construct that is used to execute a block repeatedly whenever a specified condition is met.

Usage:

```
[label :] WHILE <criteria>
  block
```

Syntax Rules

- The label must not be the same as any other label used in statements containing this one.

3.9.12. Continue Statement

A CONTINUE statement is used inside a LOOP or WHILE construct to continue with the next loop by skipping over the rest of the statements in the loop. It must be used inside a LOOP or WHILE statement.

Usage:

```
CONTINUE [label];
```

Syntax Rules

- If the label is specified, it must exist on a containing LOOP or WHILE statement.
- If no label is specified, the statement will affect the closest containing LOOP or WHILE statement.

3.9.13. Break Statement

A BREAK statement is used inside a LOOP or WHILE construct to break from the loop. It must be used inside a LOOP or WHILE statement.

Usage:

```
BREAK [label];
```

Syntax Rules

- If the label is specified, it must exist on a containing LOOP or WHILE statement.
- If no label is specified, the statement will affect the closest containing LOOP or WHILE statement.

3.9.14. Leave Statement

A LEAVE statement is used inside a compound, LOOP, or WHILE construct to leave to the specified label.

Usage:

```
LEAVE label;
```

Syntax Rules

- The label must exist on a containing compound statement, LOOP, or WHILE statement.

3.9.15. Return Statement

A Return statement gracefully exits the procedure and optionally returns a value.

Usage:

```
RETURN [expression];
```

Syntax Rules

- If an expression is specified, the procedure must have a return parameter and the value must be implicitly convertible to the expected type.
- Even if the procedure has a return value, it is not required to specify a return value in a RETURN statement.

3.9.16. Error Statement

An ERROR statement declares that the procedure has entered an error state and should abort. This statement will also roll back the current transaction, if one exists. Any valid expression can be specified after the ERROR keyword.

Usage:

```
ERROR message;
```

Example 3.18. Example Error Statement

```
ERROR 'Invalid input value: ' || nvl(Acct.GetBalance.AcctID, 'null');
```

An ERROR statement is equivalent to:

```
RAISE SQLEXCEPTION message;
```

3.9.17. Raise Statement

A RAISE statement is used to raise an exception or warning. When raising an exception, this statement will also roll back the current transaction, if one exists.

Usage:

```
RAISE [SQLWARNING] exception;
```

Where exception may be a variable reference to an exception or an exception expression.

Syntax Rules

- If SQLWARNING is specified, the exception will be sent to the client as a warning and the procedure will continue to execute.
- A null warning will be ignored. A null non-warning exception will still cause an exception to be raised.

Example 3.19. Example Raise Statement

```
RAISE SQLWARNING SQLEXCEPTION 'invalid' SQLSTATE '05000';
```

3.9.18. Exception Expression

An exception expression creates an exception that can be raised or used as a warning.

Usage:

```
SQLEXCEPTION message [SQLSTATE state [, code]] CHAIN exception
```

Syntax Rules

- Any of the values may be null;
- message and state are string expressions specifying the exception message and SQL state respectively. JBoss Data Virtualization does not yet fully comply with the ANSI SQL specification on SQL state usage, but you are allowed to set any SQL state you choose.
- code is an integer expression specifying the vendor code
- exception must be a variable reference to an exception or an exception expression and will be chained to the resulting exception as its parent.

3.10. PROCEDURES

3.10.1. Virtual Procedures

Virtual procedures are defined using the JBoss Data Virtualization procedural language (see Section 3.9.1, “Procedural Language”). A virtual procedure has zero or more input parameters, and a result set return type. Virtual procedures support the ability to execute queries and other SQL commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic.

Usage:

```
CREATE VIRTUAL PROCEDURE
block
```

The CREATE VIRTUAL PROCEDURE line indicates the beginning of the procedure. Within the body of the procedure, any valid statement may be used. See Section 3.9.1, “Procedural Language”.

There is no explicit cursoring or return statement, rather the last command statement executed in the procedure that returns a result set will be returned as the result. The output of that statement must match the expected result set and parameters of the procedure.

3.10.2. Virtual Procedure Parameters

Virtual procedures can take zero or more IN/INOUT parameters and may also have any number of OUT parameters and an optional RETURN parameter. Each input has the following information that is used during runtime processing:

- Name - The name of the input parameter.

- Datatype - The design-time type of the input parameter.
- Default value - The default value if the input parameter is not specified.
- Nullable - NO_NULLS, NULLABLE, NULLABLE_UNKNOWN; parameter is optional if nullable, and is not required to be listed when using named parameter syntax

You reference a parameter in a virtual procedure by using the fully-qualified name of the param (or less if unambiguous). For example, MySchema.MyProc.Param1.

Example 3.20. Example of Referencing an Input Parameter and Assigning an Out Parameter for 'GetBalance' Procedure

```
CREATE VIRTUAL PROCEDURE
BEGIN
  MySchema.GetBalance.RetVal = UPPER(MySchema.GetBalance.AcctID);
  SELECT Balance FROM MySchema.Accts WHERE MySchema.Accts.AccountID =
MySchema.GetBalance.AcctID;
END
```

If an INOUT parameter is not assigned any value in a procedure it will remain the value it was assigned for input. Any OUT/RETURN parameter not assigned a value will remain the as the default NULL value. The INOUT/OUT/RETURN output values are validated against the NOT NULL metadata of the parameter.

3.10.3. Example Virtual Procedures

This example is a LOOP that walks through a cursored table and uses CONTINUE and BREAK.

Example 3.21. Virtual Procedure Using LOOP, CONTINUE, BREAK

```
CREATE VIRTUAL PROCEDURE
BEGIN
  DECLARE double total;
  DECLARE integer transactions;
  LOOP ON (SELECT amt, type FROM CashTxnTable) AS txncursor
  BEGIN
    IF(txncursor.type <> 'Sale')
    BEGIN
      CONTINUE;
    END ELSE
    BEGIN
      total = (total + txncursor.amt);
      transactions = (transactions + 1);
      IF(transactions = 100)
      BEGIN
        BREAK;
      END
    END
  END
  SELECT total, (total / transactions) AS avg_transaction;
END
```

This example is uses conditional logic to determine which of two SELECT statements to execute.

Example 3.22. Virtual Procedure with Conditional SELECT

```
CREATE VIRTUAL PROCEDURE
BEGIN
  DECLARE string VARIABLES.SORTDIRECTION;
  VARIABLES.SORTDIRECTION = PartsVirtual.OrderedQtyProc.SORTMODE;
  IF ( ucase(VARIABLES.SORTDIRECTION) = 'ASC' )
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY >
PartsVirtual.OrderedQtyProc.QTYIN ORDER BY
PartsVirtual.SupplierInfo.PART_ID;
  END ELSE
  BEGIN
    SELECT * FROM PartsVirtual.SupplierInfo WHERE QUANTITY >
PartsVirtual.OrderedQtyProc.QTYIN ORDER BY
PartsVirtual.SupplierInfo.PART_ID DESC;
  END
END
```

3.10.4. Executing Virtual Procedures

You execute procedures using the SQL EXECUTE command. See Section 3.5.7, "EXECUTE Command".

If the procedure has defined inputs, you specify those in a sequential list, or using "name=value" syntax. You must use the name of the input parameter, scoped by the full procedure name if the parameter name is ambiguous in the context of other columns or variables in the procedure.

A virtual procedure call will return a result set like any SELECT, so you can use this in many places you can use a SELECT. Typically you'll use the following syntax:

```
SELECT * FROM (EXEC ...) AS x
```

3.10.5. Virtual Procedure Limitations

JBoss Data Virtualization virtual procedures can only be defined in Teiid Designer. They also cannot use IN/OUT, OUT, or RETURN parameters and may only return 1 result set.

3.10.6. Update Procedures

Views are abstractions above physical sources. They typically union or join information from multiple tables, often from multiple data sources or other views. JBoss Data Virtualization can perform update operations against views. Update commands - INSERT, UPDATE, or DELETE - against a view require logic to define how the tables and views integrated by the view are affected by each type of command. This transformation logic is invoked when an update command is issued against a view. Update procedures define the logic for how a user's update command against a view should be decomposed into the individual commands to be executed against the underlying physical sources. Similar to virtual procedures , update procedures have the ability to execute queries or other commands, define temporary tables, add data to temporary tables, walk through result sets, use loops, and use conditional logic. See Section 3.10.1, "Virtual Procedures" for more information about virtual procedures.

3.10.7. Update Procedure Processing

1. The user application submits the SQL command through one of SOAP, JDBC, or ODBC.
2. The view this SQL command is executed against is detected.
3. The correct procedure is chosen depending upon whether the command is an INSERT, UPDATE, or DELETE.
4. The procedure is executed. The procedure itself can contain SQL commands of its own which can be of different types than the command submitted by the user application that invoked the procedure.
5. Commands, as described in the procedure, are issued to the individual physical data sources or other views.
6. A value representing the number of rows changed is returned to the calling application.

3.10.8. The FOR EACH ROW Procedure

A FOR EACH ROW procedure will evaluate its block for each row of the view affected by the update statement. For UPDATE and DELETE statements this will be every row that passes the WHERE condition. For INSERT statements there will be 1 new row for each set of values from the VALUES or query expression. The rows updated is reported as this number regardless of the affect of the underlying procedure logic.

JBoss Data Virtualization FOR EACH ROW update procedures function like INSTEAD OF triggers in traditional databases. There may only be 1 FOR EACH ROW procedure for each INSERT, UPDATE, or DELETE operation against a view. FOR EACH ROW update procedures can also be used to emulate BEFORE/AFTER each row triggers while still retaining the ability to perform an inherent update. This BEFORE/AFTER trigger behavior with an inherent update can be achieved by creating an additional updatable view over the target view with update procedures of the form:

Usage:

```
FOR EACH ROW
        BEGIN ATOMIC
          ...
        END
```

The BEGIN and END keywords are used to denote block boundaries. Within the body of the procedure, any valid statement may be used. See Section 3.9.1, “Procedural Language”.

NOTE

Use of the ATOMIC keyword is currently optional for backward compatibility, but unlike a normal block, the default for INSTEAD OF is atomic.

3.10.9. Special Variables for Update Procedures

You can use a number of special variables when defining your update procedure.

NEW

Every attribute in the view whose UPDATE and INSERT transformations you are defining has an equivalent variable named NEW.<column_name>

When an INSERT or an UPDATE command is executed against the view, these variables are initialized to the values in the INSERT VALUES clause or the UPDATE SET clause respectively.

In an UPDATE procedure, the default value of these variables, if they are not set by the command, is the old value. In an INSERT procedure, the default value of these variables is the default value of the virtual table attributes. See CHANGING variables for distinguishing defaults from passed values.

OLD

Every attribute in the view whose UPDATE and DELETE transformations you are defining has an equivalent variable named OLD.<column_name>

When a DELETE or UPDATE command is executed against the view, these variables are initialized to the current values of the row being deleted or updated respectively.

CHANGING

Every attribute in the view whose UPDATE and INSERT transformations you are defining has an equivalent variable named CHANGING.<column_name>

When an INSERT or an UPDATE command is executed against the view, these variables are initialized to `true` or `false` depending on whether the INPUT variable was set by the command. A CHANGING variable is commonly used to differentiate between a default insert value and one specified in the user query.

For example, for a view with columns A, B, C:

If User Executes...	Then...
`INSERT INTO VT (A, B) VALUES (0, 1)`	CHANGING.A = true, CHANGING.B = true, CHANGING.C = false
`UPDATE VT SET C = 2`	CHANGING.A = false, CHANGING.B = false, CHANGING.C = true

3.10.10. Example Update Procedures

For example, for a view with columns A, B, C:

Example 3.23. Sample DELETE Procedure

```
FOR EACH ROW
BEGIN
        DELETE FROM X WHERE Y = OLD.A;
        DELETE FROM Z WHERE Y = OLD.A; // cascade the delete
END
```

Example 3.24. Sample UPDATE Procedure

```
FOR EACH ROW
BEGIN
```

```
        IF (CHANGING.B)
        BEGIN
                    UPDATE Z SET Y = NEW.B WHERE Y = OLD.B;
        END
    END
```

CHAPTER 4. DATA TYPES

4.1. SUPPORTED TYPES

JBoss Data Virtualization supports a core set of runtime types. Runtime types can be different from semantic types defined in type fields at design time. The runtime type can also be specified at design time or it will be automatically chosen as the closest base type to the semantic type.

Table 4.1. JBoss Data Virtualization Runtime Types

Type	Description	Java Runtime Class	JDBC Type	ODBC Type
string or varchar	variable length character string with a maximum length of 4000. Note that the length cannot be explicitly set with the type declaration, e.g. varchar(100) is invalid.	java.lang.String	VARCHAR	VARCHAR
varbinary	variable length binary string with a maximum length of 8192. Note that the length cannot be explicitly set with the type declaration, e.g. varbinary(100) is invalid.	byte[] [a]	VARBINARY	VARBINARY
char	a single Unicode character	java.lang.Character	CHAR	CHAR
boolean	a single bit, or Boolean, that can be true, false, or null (unknown)	java.lang.Boolean	BIT	SMALLINT
byte or tinyint	numeric, integral type, signed 8-bit	java.lang.Byte	TINYINT	SMALLINT
short or smallint	numeric, integral type, signed 16-bit	java.lang.Short	SMALLINT	SMALLINT
integer or serial	numeric, integral type, signed 32-bit. The serial type also implies not null and has an auto-incrementing value that starts at 1. Serial types are not automatically UNIQUE.	java.lang.Integer	INTEGER	INTEGER
long or bigint	numeric, integral type, signed 64-bit	java.lang.Long	BIGINT	NUMERIC

Type	Description	Java Runtime Class	JDBC Type	ODBC Type
biginteger	numeric, integral type, arbitrary precision of up to 1000 digits	java.math.BigInteger	NUMERIC	NUMERIC
float or real	numeric, floating point type, 32-bit IEEE 754 floating-point numbers	java.lang.Float	REAL	FLOAT
double	numeric, floating point type, 64-bit IEEE 754 floating-point numbers	java.lang.Double	DOUBLE	DOUBLE
bigdecimal or decimal	numeric, floating point type, arbitrary precision of up to 1000 digits. Note that the precision and scale cannot be explicitly set with the type literal, e.g. decimal(38, 2).	java.math.BigDecimal	NUMERIC	NUMERIC
date	datetime, representing a single day (year, month, day)	java.sql.Date	DATE	DATE
time	datetime, representing a single time (hours, minutes, seconds, milliseconds)	java.sql.Time	TIME	TIME
timestamp	datetime, representing a single date and time (year, month, day, hours, minutes, seconds, milliseconds, nanoseconds)	java.sql.Timestamp	TIMESTAMP	TIMESTAMP
object	any arbitrary Java object, must implement java.lang.Serializable	Any	JAVA_OBJECT	VARCHAR
blob	binary large object, representing a stream of bytes	java.sql.Blob [b]	BLOB	VARCHAR
clob	character large object, representing a stream of characters	java.sql.Clob [c]	CLOB	VARCHAR
xml	XML document	java.sql.SQLXML [d]	JAVA_OBJECT	VARCHAR

Type	Description	Java Runtime Class	JDBC Type	ODBC Type

[a] The runtime type is org.teiid.core.types.BinaryType. Translators will need to explicitly handle BinaryType values. UDFs will instead have a byte[] value passed.

[b] The concrete type is expected to be org.teiid.core.types.BlobType

[c] The concrete type is expected to be org.teiid.core.types.ClobType

[d] The concrete type is expected to be org.teiid.core.types.XMLType

4.2. TYPE CONVERSIONS

Data types may be converted from one form to another either explicitly or implicitly. Implicit conversions automatically occur in criteria and expressions to ease development. Explicit data type conversions require the use of the **CONVERT** function or **CAST** keyword.

Type Conversion Considerations

- Any type may be implicitly converted to the OBJECT type.
- The OBJECT type may be explicitly converted to any other type.
- The **NULL** value may be converted to any type.
- Any valid implicit conversion is also a valid explicit conversion.
- Situations involving literal values that would normally require explicit conversions may have the explicit conversion applied implicitly if no loss of information occurs.
- When JBoss Data Virtualization detects that an explicit conversion can not be applied implicitly in criteria, the criteria will be treated as false. For example:

  ```
  SELECT * FROM my.table WHERE created_by = 'not a date'
  ```

  Given that created_by is typed as date, rather than converting **'not a date'** to a date value, the criteria will remain as a string comparison and therefore be false.
- Explicit conversions that are not allowed between two types will result in an exception before execution. Allowed explicit conversions may still fail during processing if the runtime values are not actually convertible.
- **WARNING**

  The JBoss Data Virtualization conversions of float/double/bigdecimal/timestamp to string rely on the JDBC/Java defined output formats. Pushdown behavior attempts to mimic these results, but may vary depending upon the actual source type and conversion logic. Care must be taken to not assume the string form in criteria or other places where a variation may cause different results.

Table 4.2. Type Conversions

Source Type	Valid Implicit Target Types	Valid Explicit Target Types
string	clob	char, boolean, byte, short, integer, long, biginteger, float, double, bigdecimal, xml [a]
char	string	
boolean	string, byte, short, integer, long, biginteger, float, double, bigdecimal	
byte	string, short, integer, long, biginteger, float, double, bigdecimal	boolean
short	string, integer, long, biginteger, float, double, bigdecimal	boolean, byte
integer	string, long, biginteger, double, bigdecimal	boolean, byte, short, float
long	string, biginteger, bigdecimal	boolean, byte, short, integer, float, double
biginteger	string, bigdecimal	boolean, byte, short, integer, long, float, double
bigdecimal	string	boolean, byte, short, integer, long, biginteger, float, double
date	string, timestamp	
time	string, timestamp	
timestamp	string	date, time
clob		string
xml		string [b]

[a] string to xml is equivlant to XMLPARSE(DOCUMENT exp) - see Section 3.4.14, “XML Functions”.

[b] xml to string is equivalent to XMLSERIALIZE(exp AS STRING) - see Section 3.4.14, “XML Functions”.

4.3. CONVERSION OF STRING LITERALS

JBoss Data Virtualization automatically converts string literals within an SQL statement to their implied types. This typically occurs in a criteria comparison where an expression with a different data type is compared to a literal string:

```
SELECT * FROM my.table WHERE created_by = '2003-01-02'
```

Here if the created_by column has the data type of date, JBoss Data Virtualization automatically converts the string literal to a date data type as well.

4.4. CONVERTING TO BOOLEAN

JBoss Data Virtualization can automatically convert literal strings and numeric type values to Boolean values as follows:

Type	Literal Value	Boolean Value
String	'false'	false
	'unknown'	null
	other	true
Numeric	0	false
	other	true

4.5. DATE AND TIME CONVERSIONS

JBoss Data Virtualization can implicitly convert properly formatted literal strings to their associated date-related data types as follows:

String Literal Format	Possible Implicit Conversion Type
yyyy-mm-dd	DATE
hh:mm:ss	TIME
yyyy-mm-dd hh:mm:ss.[fff...]	TIMESTAMP

The formats above are those expected by the JDBC date types. To use other formats see the functions **PARSEDATE**, **PARSETIME**, **PARSETIMESTAMP**.

4.6. ESCAPED LITERAL SYNTAX

Rather than relying on implicit conversion, data type values may be expressed directly in SQL using escape syntax to define the type. Note that the supplied string value must match the expected format exactly or an exception will occur.

Table 4.3. Escaped Literal Syntax

Data type	Escaped Syntax
DATE	{d 'yyyy-mm-dd'}
TIME	{t 'hh-mm-ss'}
TIMESTAMP	{ts 'yyyy-mm-dd hh:mm:ss.[fff...]'}

CHAPTER 5. UPDATABLE VIEWS

5.1. UPDATABLE VIEWS

Any view may be marked as updatable. In many circumstances the view definition may allow the view to be inherently updatable without the need to manually define handling of INSERT/UPDATE/DELETE operations.

An inherently updatable view cannot be defined with a query that has:

- A set operation (INTERSECT, EXCEPT, UNION).
- SELECT DISTINCT
- Aggregation (aggregate functions, GROUP BY, HAVING)
- A LIMIT clause

A UNION ALL can define an inherently updatable view only if each of the UNION branches is itself inherently updatable. A view defined by a UNION ALL can support inherent INSERTs if it is a partitioned union and the INSERT specifies values that belong to a single partition. Refer to Partitioned Union.

Any view column that is not mapped directly to a column is not updatable and cannot be targeted by an UPDATE set clause or be an INSERT column.

If a view is defined by a join query or has a WITH clause it may still be inherently updatable. However in these situations there are further restrictions and the resulting query plan may execute multiple statements. For a non-simple query to be updatable, it is required:

- An INSERT/UPDATE can only modify a single key-preserved table.
- To allow DELETE operations there must be only a single key-preserved table.

If the default handling is not available or you wish to have an alternative implementation of an INSERT/UPDATE/DELETE, then you may use update procedures (see Section 3.10.6, “Update Procedures”) to define procedures to handle the respective operations.

5.2. KEY-PRESERVED TABLE

A key-preserved table has a primary or unique key that would remain unique if it were projected into the result of the query. Note that it is not actually required for a view to reference the key columns in the SELECT clause. The query engine can detect a key preserved table by analyzing the join structure. The engine will ensure that a join of a key-preserved table must be against one of its foreign keys.

CHAPTER 6. TRANSACTION SUPPORT

6.1. TRANSACTION SUPPORT

JBoss Data Virtualization uses XA transactions for participating in global transactions and for demarcating its local and command scoped transactions. Refer to the Red Hat JBoss Data Virtualization *Development Guide Volume 1: Client Development*for more information about the transaction subsystem.

Table 6.1. JBoss Data Virtualization Transaction Scopes

Scope	Description
Command	Treats the user command as if all source commands are executed within the scope of the same transaction. The AutoCommitTxn execution property controls the behavior of command level transactions.
Local	The transaction boundary is local defined by a single client session.
Global	JBoss Data Virtualization participates in a global transaction as an XA Resource.

The default transaction isolation level for JBoss Data Virtualization is READ_COMMITTED.

6.2. AUTOCOMMITTXN EXECUTION PROPERTY

Since user level commands may execute multiple source commands, users can specify the AutoCommitTxn execution property to control the transactional behavior of a user command when not in a local or global transaction.

Table 6.2. AutoCommitTxn Settings

Setting	Description
OFF	Do not wrap each command in a transaction. Individual source commands may commit or rollback regardless of the success or failure of the overall command.
ON	Wrap each command in a transaction. This mode is the safest, but may introduce performance overhead.
DETECT	This is the default setting. Will automatically wrap commands in a transaction, but only if the command seems to be transactionally unsafe.

The concept of command safety with respect to a transaction is determined by Red Hat JBoss Data Virtualization based upon command type, the transaction isolation level, and available metadata. A wrapping transaction is not needed if any of the following is true:

- A user command is fully pushed to the source.
- The user command is a SELECT (including XML) and the transaction isolation is not REPEATABLE_READ nor SERIALIZABLE.

- The user command is a stored procedure and the transaction isolation is not REPEATABLE_READ nor SERIALIZABLE and the updating model count is zero.

The update count may be set on all procedures as part of the procedure metadata in the model.

6.3. UPDATING MODEL COUNT

The term "updating model count" refers to the number of times any model is updated during the execution of a command. It is used to determine whether a transaction, of any scope, is required to safely execute the command.

Table 6.3. Updating Model Count Settings

Count	Description
0	No updates are performed by this command.
1	Indicates that only one model is updated by this command (and its subcommands). Also the success or failure of that update corresponds to the success or failure of the command. It should not be possible for the update to succeed while the command fails. Execution is not considered transactionally unsafe.
*	Any number greater than 1 indicates that execution is transactionally unsafe and an XA transaction will be required.

6.4. JDBC API FUNCTIONALITY

The transaction scopes in Section 6.1, “Transaction Support” map to the following JDBC modes:

Command

Connection autoCommit property set to true.

Local

Connection autoCommit property set to false. The transaction is committed by setting autoCommit to true or calling `java.sql.Connection.commit`. The transaction can be rolled back by a call to `java.sql.Connection.rollback`.

Global

The XAResource interface provided by an XAConnection is used to control the transaction. Note that XAConnections are available only if JBoss Data Virtualization is consumed through its XADataSource, `org.teiid.jdbc.TeiidDataSource`. JEE containers or data access APIs typically control XA transactions on behalf of application code.

6.5. J2EE USAGE MODELS

J2EE provides three ways to manage transactions for beans:

Client-Controlled

The client of a bean begins and ends a transaction explicitly.

Bean-Managed

The bean itself begins and ends a transaction explicitly.

Container-Managed

The application server container begins and ends a transaction automatically.

In any of these cases, transactions may be either local or XA transactions, depending on how the code and descriptors are written. Some kinds of beans (stateful session beans and entity beans) are not required by the spec to support non-transactional sources, although the spec does allow an application server to optionally support this with the caution that this is not portable or predictable. Generally speaking, to support most typical EJB activities in a portable fashion requires some kind of transaction support.

6.6. TRANSACTIONAL BEHAVIOR WITH JBOSS DATA SOURCE TYPES

JBoss Enterprise Application Platform allows creation of different types of data sources, based on their transactional capabilities. The type of data source you create for your VDB's sources also dictates if that data source will be participating the distributed transaction or not, irrespective of the transaction scope you selected from above. Here are different types of data sources:

- xa-datasource: Capable of participating in the distributed transaction using XA. This is the recommended type be used with any JBoss Data Virtualization sources.
- local-datasource: Does not participate in XA, unless this is the *only* local-datasource participating among other xa-datasources in the current distributed transaction. This technique is called last commit optimization. However, if you have more than one local datasource participating in a transaction, the transaction manager will throw an exception: *"Could not enlist in transaction on entering meta-aware object!."*
- no-tx-datasource: Does not participate in distributed transaction at all. In the scope of a JBoss Data Virtualization command over multiple sources, you can include this type of datasource in the same distributed transaction context, however this source will not be subject to any transactional participation. Any changes done on this source as part of the transaction scope, cannot be rolled back.

For example, if you have three different sources A, B, C being used in JBoss Data Virtualization, here are some variations on how they behave with different types of data sources. The suffixes "xa", "local", "no-tx" define different type of sources used.

- A-xa B-xa, C-xa : Can participate in all transactional scopes. No restrictions.
- A-xa, B-xa, c-local: Can participate in all transactional scopes. Note that there is only one single source, "local". It is assumed that, in the Global scope, any third party datasource other than JBoss Data Virtualization datasource is also XA.
- A-xa, B-xa, C-no-tx : Can participate in all transactional scopes. Note "C" is not bound by any transactional contract. A and B are the only participants in the XA transaction.
- A-xa, B-local, C-no-tx : Can participate in all transactional scopes. Note "C" is not bound by any transactional contract, and there is only a single "local" source.
- If any two or more sources are "local" : They can only participate in Command mode with "autoCommitTxn=OFF". Otherwise they will end with an exception and the message "Could not enlist in transaction on entering meta-aware object!;" because it is not possible to do a XA

transaction with "local" datasources.

- A-no-tx, B-no-tx, C-no-tx : Can participate in all transaction scopes, but none of the sources will be bound by transactional terms. This is equivalent to not using transactions or setting Command mode with "autoCommitTxn=OFF".

IMPORTANT

Teiid Designer creates a "local" data source by default. This is not optimal for XA transactions. To create XA datasources, use the Management Console. You can find examples in the `EAP_HOME/docs/teiid/datasources` directory.

If your datasource is not XA, and not the only local source and cannot use "no-tx", then you can look into extending the source to implement the compensating XA implementation. Define your own resource manager for your source and manage the transaction the way you want it to behave. Note that this could be complicated if your source natively does not support the distributed XA protocol.

In summary:

- Use XA datasource if possible
- Use no-tx datasource if applicable
- Use autoCommitTxn = OFF, and let go distributed transactions, though not recommended
- Write a compensating XA based implementation.

Table 6.4. Data Virtualization Transaction Participation

Teiid-Tx-Scope	XA source	Local Source	No-Tx Source
Local	always	Only If Single Source	never
Global	always	Only If Single Source	never
Auto-commit=true, AutoCommitTxn=ON	always	Only If Single Source	never
Auto-commit=true, AutoCommitTxn=OFF	never	never	never
Auto-commit=true, AutoCommitTxn=DETECT	always	Only If Single Source	never

6.7. LIMITATIONS

- The client setting of transaction isolation level is not propagated to the connectors. The transaction isolation level can be set on each XA connector, however this isolation level is fixed and cannot be changed at runtime for specific connections/commands.

CHAPTER 7. DATA ROLES

7.1. DATA ROLES

Data roles, also called entitlements, are sets of permissions defined per VDB that dictate data access (create, read, update, delete). Data roles use a fine-grained permission system that JBoss Data Virtualization will enforce at runtime and provide audit log entries for access violations. Refer to the Administration and Configuration Guide and Development Guide: Server Development for more information about Logging and Custom Logging.

Prior to applying data roles, you should consider restricting source system access through the fundamental design of your VDB. Foremost, JBoss Data Virtualization can only access source entries that are represented in imported metadata. You should narrow imported metadata to only what is necessary for use by your VDB. When using Teiid Designer, you may then go further and modify the imported metadata at a granular level to remove specific columns or indicate tables that are not to be updated, etc.

If data role validation is enabled and data roles are defined in a VDB, then access permissions will be enforced by the JBoss Data Virtualization Server. The use of data roles may be disabled system wide using the setting for the **`teiid`** subsystem policy-decider-module. Data roles also have built-in system functions (see Section 3.4.17, "Security Functions") that can be used for row-based and other authorization checks.

The **`hasRole`** system function will return true if the current user has the given data role. The **`hasRole`** function can be used in procedure or view definitions to allow for a more dynamic application of security - which allows for things such as value masking or row level security.

NOTE

See the Security Guide for details on using an alternative authorization scheme.

WARNING

Data roles are only checked if present in a VDB. A VDB deployed without data roles can be used by any authenticated user.

7.2. ROLE MAPPING

Each JBoss Data Virtualization data role can be mapped to any number of container roles or any authenticated user. Control role membership through whatever system the JBoss Data Virtualization security domain login modules are associated with.

It is possible for a user to have any number of container roles, which in turn imply a subset of JBoss Data Virtualization data roles. Each applicable JBoss Data Virtualization data role contributes cumulatively to the permissions of the user. No one role supersedes or negates the permissions of the other data roles.

7.3. PERMISSIONS

7.3.1. User Query Permissions

CREATE, READ, UPDATE, DELETE (CRUD) permissions can be set for any resource path in a VDB. A resource path can be as specific as the fully qualified name of a column or as general a top level model (schema) name. Permissions granted to a particular path apply to it and any resource paths that share the same partial name. For example, granting read to "model" will also grant read to "model.table", "model.table.column", etc. Allowing or denying a particular action is determined by searching for permissions from the most to least specific resource paths. The first permission found with a specific allow or deny will be used. Thus it is possible to set very general permissions at high-level resource path names and to override only as necessary at more specific resource paths.

Permission grants are only needed for resources that a role needs access to. Permissions are also only applied to the columns/tables/procedures in the user query - not to every resource accessed transitively through view and procedure definitions. It is important therefore to ensure that permission grants are applied consistently across models that access the same resources.

WARNING

Non-visible models are accessible by user queries. To restrict user access at a model level, at least one data role should be created to enable data role checking. In turn that role can be mapped to any authenticated user and should not grant permissions to models that should be inaccessible.

Permissions are not applicable to the SYS and pg_catalog schemas. These metadata reporting schemas are always accessible regardless of the user. The SYSADMIN schema however may need permissions as applicable.

7.3.2. Assigning Permissions

To process a *SELECT* statement or a stored procedure execution, the user account requires the following access rights:

1. *READ* - on the Table(s) being accessed or the procedure being called.
2. *READ* - on every column referenced.

To process an *INSERT* statement, the user account requires the following access rights:

1. *CREATE* - on the Table being inserted into.
2. *CREATE* - on every column being inserted on that Table.

To process an *UPDATE* statement, the user account requires the following access rights:

1. *UPDATE* - on the Table being updated.
2. *UPDATE* - on every column being updated on that Table.
3. *READ* - on every column referenced in the criteria.

To process a *DELETE* statement, the user account requires the following access rights:

1. *DELETE* - on the Table being deleted.
2. *READ* - on every column referenced in the criteria.

To process a *EXEC/CALL* statement, the user account requires the following access rights:

1. *EXECUTE (or READ)*- on the Procedure being executed.

To process any function, the user account requires the following access rights:

1. *EXECUTE (or READ)*- on the Function being called.

To process any *ALTER* or *CREATE TRIGGER* statement, the user account requires the following access rights:

1. *ALTER* - on the view or procedure that is effected. INSTEAD OF Triggers (update procedures) are not yet treated as full schema objects and are instead treated as attributes of the view.

To process any *OBJECTTABLE* function, the user account requires the following access rights:

1. *LANGUAGE* - specifying the language name that is allowed.

To process any statement against a JBoss Data Virtualization temporary table requires the following access rights:

1. allow-create-temporary-tables attribute on any applicable role
2. *CREATE* - against the target source/schema if defining a FOREIGN temporary table.

7.3.3. Row and Column-Based Security Conditions

Although specified in a similar way to user query CRUD permissions, row-based and column-based permissions may be used together or separately to control at a more granular and consistent level the data returned to users.

7.3.4. Row-Based Security Conditions

A permission against a fully qualified table/view/procedure may also specify a condition. Unlike the allow actions defined above, a condition is always applied - not only at the user query level. The condition can be any valid SQL referencing the columns of the table/view/procedure. The condition will act as a row-based filter and as a checked constraint for insert/update operations.

7.3.5. Applying Row-Based Security Conditions

A condition is applied conjunctively to UPDATE/DELETE/SELECT WHERE clauses against the affected resource. Those queries will therefore only ever be effective against the subset of rows that pass the condition, i.e. "SELECT * FROM TBL WHERE something **AND condition** ". The condition will be present regardless of how the table/view is used in the query, whether via a union, join, etc.

Inserts and updates against physical tables affected by a condition are further validated so that the insert/change values must pass the condition (evaluate to true) for the insert/update to succeed - this is effectively the same as an SQL constraint. This will happen for all styles of insert/update - insert with query expression, bulk insert/update, etc. Inserts/updates against views are not checked with regards to the constraint. You can disable the insert/update constraint check by setting the condition constraint flag to false. This is typically only needed in circumstances when the condition cannot always be evaluated. However disabling the condition as a constraint drops the condition from consideration when logically evaluating the constraint. Any other condition constraints will still be evaluated.

Across multiple applicable roles, if more than one condition applies to the same resource, the conditions will be accumulated disjunctively via OR, i.e. "(condition1) **OR** (condition2) …". Therefore granting a permission with the condition "true" will allow users in that role to see all rows of the given resource.

7.3.6. Considerations When Using Conditions

Non-pushdown conditions may adversely impact performance, since their evaluation may inhibit pushdown of query constructs on top of the affected resource. Multiple conditions against the same resource should generally be avoided as any non-pushdown condition will cause the entire OR of conditions to not be pushed down. In some circumstances the insertion of permission conditions may require that the plan be altered with the addition of an inline view, which can result in adverse performance against sources that do not support inline views.

Pushdown of multi-row insert/update operations will be inhibited since the condition must be checked for each row.

In addition to managing permission conditions on a per-role basis, another approach is to add condition permissions would in an any authenticated role such that the conditions are generalized for all users/roles using the **`hasRole`** , **`user`** , and other such security functions. The advantage of the latter approach is that there is effectively a static row-based policy in effect such that all query plans can still be shared between users.

Handling of null values is up to the implementer of the data role and may require ISNULL checks to ensure that null values are allowed when a column is nullable.

7.3.7. Limitations to Using Conditions

- Conditions on source tables that act as check constraints must currently not contain correlated subqueries.
- Conditions may not contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.
- **NOTE**

  Row-based filter conditions are enforced even for materialized view loads.

  You should ensure that tables consumed to produce materialized views do not have row-based filter conditions on them that could affect the materialized view results.

7.3.8. Column Masking

A permission against a fully qualified table/view/procedure column may also specify a mask and optionally a condition. When the query is submitted the roles are consulted and the relevant mask/condition information are combined to form a searched case expression to mask the values that would have been returned by the access. Unlike the CRUD allow actions defined above, the resulting masking effect is always applied - not only at the user query level. The condition and expression can be any valid SQL referencing the columns of the table/view/procedure. Procedure result set columns may be referenced as proc.col.

7.3.9. Applying Column Masking

Column masking is applied only against SELECTs. Column masking is applied logically after the affect of row based security. However since both views and source tables may have row and column based security, the actual view level masking may take place on top of source level masking. If the condition is specified along with the mask, then the effective mask expression effects only a subset of the rows: "CASE WHEN condition THEN mask ELSE column". Otherwise the condition is assumed to be TRUE, meaning that the mask applies to all rows.

If multiple roles specify a mask against a column, the mask order argument will determine their precedence from highest to lowest as part of a larger searched case expression. For example a mask with the default order of 0 and a mask with an order of 1 would be combined as "CASE WHEN condition1 THEN mask1 WHEN condition0 THEN mask0 ELSE column".

7.3.10. Column Masking Considerations

Non-pushdown masking conditions/expressions may adversely impact performance, since their evaluation may inhibit pushdown of query constructs on top of the affected resource. In some circumstances the insertion of masking may require that the plan be altered with the addition of an inline view, which can result in adverse performance against sources that do not support inline views.

In addition to managing masking on a per-role basis with the use of the order value, another approach is to specify masking in a single any authenticated role such that the conditions/expressions are generalized for all users/roles using the `hasRole`, `user`, and other such security functions. The advantage of the latter approach is that there is effectively a static masking policy in effect such that all query plans can still be shared between users.

7.3.11. Column Masking Limitations

- In the event that two masks have the same order value, it is not well defined what order they are applied in.
- Masks or their conditions may not contain aggregate or windowed functions.
- Tables and procedures referenced via subqueries will still have row-based filters and column masking applied to them.
- **NOTE**

  Masking is enforced even for materialized view loads.

  You should ensure that tables consumed to produce materialized views do not have masking on them that could affect the materialized view results.

7.4. DATA ROLE DEFINITION

7.4.1. Data Role Definition

Data roles are defined inside the **META-INF/vdb.xml** file of the VDB archive if you used the Teiid Designer. The **vdb.xml** file is checked against the **vdb-deployer.xsd** schema file found in the ***EAP_HOME*/docs/teiid/schema** directory.

7.4.2. Data Role Definition Example

Consider the scenario in which a VDB defines a table "TableA" in schema "modelName" with columns (column1, column2) - note that the column types do not matter.

We wish to define three roles "RoleA", "RoleB", "RoleC" with the following permissions:

1. RoleA has permissions to read, write access to TableA, but can not delete.
2. RoleB has no permissions that allow access to TableA
3. RoleC has permissions that only allow read access to TableA.column1

Example 7.1. vdb.xml defining RoleA, RoleB, and RoleC

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-
jndi-name="java:myDS" />
    </model>

    <data-role name="RoleA">
        <description>Allow all, except Delete</description>

        <permission>
            <resource-name>modelName.TableA</resource-name>
            <allow-create>true</allow-create>
            <allow-read>true</allow-read>
            <allow-update>true</allow-update>
        </permission>

        <mapped-role-name>role1</mapped-role-name>

    </data-role>

    <data-role name="RoleC">
        <description>Allow read only</description>

        <permission>
            <resource-name>modelName.TableA</resource-name>
            <allow-read>true</allow-read>
        </permission>

        <permission>
            <resource-name>modelName.TableA.colum2</resource-name>
            <allow-read>false</allow-read>
        </permission>

        <mapped-role-name>role2</mapped-role-name>
    </data-role>
```

```
</vdb>
```

The above XML defined two data roles, "RoleA" which allows everything except delete on the table, "RoleC" that allows only read operation on the table. Since JBoss Data Virtualization uses deny by default, there is no explicit data-role entry needed for "RoleB". Note that explicit column permissions are not needed for RoleA, since the parent resource path, modelName.TableA, permissions still apply. RoleC however must explicitly disallow read to column2.

The "mapped-role-name" defines the container JAAS roles that are assigned the data role. For assigning roles to your users in the JBoss EAP, see the instructions for the selected Login Module. See the Administrator Guide for configuring Login Modules.

7.4.3. Data Role Definition Example: Additional Attributes

You may also choose to allow any authenticated user to have a data role by setting the any-authenticated attribute value to true on data-role element.

The "allow-create-temporary-tables" data-role boolean attribute is used to explicitly enable or disable temporary table usage for the role. If it is left unspecified, then the value will be defaulted to false.

Example 7.2. Temp Table Role for Any Authenticated

```
<data-role name="role" any-authenticated="true" allow-create-temporary-
tables="true">
     <description>Temp Table Role for Any Authenticated</description>

     <permission>
         ...
     </permission>

</data-role>
```

7.4.4. Data Role Definition Example: Language Access

The following shows a vdb xml that allows the use of the javascript language. The allowed-languages property enables the languages use for any purpose in the vdb, while the allow-language permission allows the language to be used by users with RoleA.

Example 7.3. vdb.xml allowing JavaScript access

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <property name="allowed-languages" value="javascript"/>

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-
jndi-name="java:myDS" />
    </model>

    <data-role name="RoleA">
        <description>Read and javascript access.</description>
        <permission>
            <resource-name>modelName</resource-name>
            <allow-read>true</allow-read>
        </permission>

        <permission>
            <resource-name>javascript</resource-name>
            <allow-language>true</allow-language>
        </permission>

        <mapped-role-name>role1</mapped-role-name>

    </data-role>

</vdb>
```

7.4.5. Data Role Definition Example: Row-Based Security

The following shows a VDB XML definition utilizing a condition to restrict access. The condition acts as both a filter and constraint. Even though RoleA opens up read/insert access to modelName.tblName, the base-role condition will ensure that only values of column1 matching the current user can be read or inserted. Note that here the constraint enforcement has been disabled.

Example 7.4. vdb.xml allowing conditional access

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-
jndi-name="java:myDS" />
    </model>

    <data-role name="base-role" any-authenticated="true">
        <description>Conditional access</description>

        <permission>
            <resource-name>modelName.tblName</resource-name>
            <condition constraint="false">column1=user()</condition>
        </permission>

    </data-role>

    <data-role name="RoleA">
        <description>Read/Insert access.</description>

        <permission>
            <resource-name>modelName.tblName</resource-name>
            <allow-read>true</allow-read>
            <allow-create>true</allow-create>
        </permission>

        <mapped-role-name>role1</mapped-role-name>
    </data-role>

</vdb>
```

7.4.6. Data Role Definition Example: Column Masking

The following shows VDB XML utilizing column masking. Here the RoleA column1 mask takes precedence over the base-role mask, but only for a subset of the rows as specified by the condition. For users without RoleA, access to column1 will effectively be replaced with "CASE WHEN column1=user() THEN column1 END", while for users with RoleA, access to column1 will effectively be replaced with "CASE WHEN column2='x' THEN column1 WHEN TRUE THEN CASE WHEN column1=user() THEN column1 END END".

Example 7.5. vdb.xml with column masking

```
<?xml version="1.0" encoding="UTF-8"?>
<vdb name="sample" version="1">

    <model name="modelName">
        <source name="source-name" translator-name="oracle" connection-
jndi-name="java:myDS" />
    </model>

    <data-role name="base-role" any-authenticated="true">
        <description>Masking</description>

        <permission>
            <resource-name>modelName.tblName.column1</resource-name>
            <mask>CASE WHEN column1=user() THEN column1 END</mask>
        </permission>

    </data-role>

    <data-role name="RoleA">
        <description>Read/Insert access.</description>

        <permission>
            <resource-name>modelName.tblName</resource-name>
            <allow-read>true</allow-read>
            <allow-create>true</allow-create>
        </permission>

        <permission>
            <resource-name>modelName.tblName.column1</resource-name>
            <condition>column2='x'</condition>
            <mask order="1">column1</mask>
        </permission>

        <mapped-role-name>role1</mapped-role-name>

    </data-role>

</vdb>
```

CHAPTER 8. SYSTEM SCHEMAS AND PROCEDURES

8.1. SYSTEM SCHEMAS

The built-in SYS and SYSADMIN schemas provide metadata tables and procedures against the current virtual database.

8.2. VDB METADATA

SYSADMIN.VDBResources

This table provides the current VDB contents.

Column Name	Type	Description
resourcePath	string	The path to the contents.
contents	blob	The contents as a blob.

SYS.VirtualDatabases

This table supplies information about the currently connected virtual database, of which there is always exactly one (in the context of a connection).

Column Name	Type	Description
Name	string	The name of the VDB
Version	string	The version of the VDB

SYS.Schemas

This table supplies information about all the schemas in the virtual database, including the system schema itself (System).

Column Name	Type	Description
VDBName	string	VDB name
Name	string	Schema name
IsPhysical	boolean	True if this represents a source
UID	string	Unique ID
OID	integer	Unique ID

Column Name	Type	Description
Description	string	Description
PrimaryMetamodelURI	string	URI for the primary metamodel describing the model used for this schema

SYS.Properties

This table supplies user-defined properties on all objects based on metamodel extensions. Normally, this table is empty if no metamodel extensions are being used.

Column Name	Type	Description
Name	string	Extension property name
Value	string	Extension property value
UID	string	Key unique ID
OID	integer	Unique ID

WARNING

The OID column is no longer used on system tables. Use UID instead.

8.3. TABLE METADATA

SYS.Tables

This table supplies information about all the groups (tables, views, documents, etc) in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Short group name

Column Name	Type	Description
Type	string	Table type (Table, View, Document, ...)
NameInSource	string	Name of this group in the source
IsPhysical	boolean	True if this is a source table
SupportsUpdates	boolean	True if group can be updated
UID	string	Group unique ID
OID	integer	Unique ID
Cardinality	integer	Approximate number of rows in the group
Description	string	Description
IsSystem	boolean	True if in system table

SYS.Columns

This table supplies information about all the elements (columns, tags, attributes, etc) in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
TableName	string	Table name
Name	string	Element name (not qualified)
Position	integer	Position in group (1-based)
NameInSource	string	Name of element in source
DataType	string	Data Virtualization runtime data type name
Scale	integer	Number of digits after the decimal point

Column Name	Type	Description
ElementLength	integer	Element length (mostly used for strings)
sLengthFixed	boolean	Whether the length is fixed or variable
SupportsSelect	boolean	Element can be used in SELECT
SupportsUpdates	boolean	Values can be inserted or updated in the element
IsCaseSensitive	boolean	Element is case-sensitive
IsSigned	boolean	Element is signed numeric value
IsCurrency	boolean	Element represents monetary value
IsAutoIncremented	boolean	Element is auto-incremented in the source
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
MinRange	string	Minimum value
MaxRange	string	Maximum value
DistinctCount	integer	Distinct value count, -1 can indicate unknown
NullCount	integer	Null value count, -1 can indicate unknown
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", "Unsearchable"
Format	string	Format of string value
DefaultValue	string	Default value
JavaClass	string	Java class that will be returned
Precision	integer	Number of digits in numeric value
CharOctetLength	integer	Measure of return value size

Column Name	Type	Description
Radix	integer	Radix for numeric values
GroupUpperName	string	Upper-case full group name
UpperName	string	Upper-case element name
UID	string	Element unique ID
OID	integer	Unique ID
Description	string	Description

SYS.Keys

This table supplies information about primary, foreign, and unique keys.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Table Name	string	Table name
Name	string	Key name
Description	string	Description
NameInSource	string	Name of key in source system
Type	string	Type of key: "Primary", "Foreign", "Unique", etc
IsIndexed	boolean	True if key is indexed
RefKeyUID	string	Referenced key UID (if foreign key)
UID	string	Key unique ID
OID	integer	Unique ID

SYS.KeyColumns

This table supplies information about the columns referenced by a key.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
TableName	string	Table name
Name	string	Element name
KeyName	string	Key name
KeyType	string	Key type: "Primary", "Foreign", "Unique", etc
RefKeyUID	string	Referenced key UID
UID	string	Key UID
OID	integer	Unique ID
Position	integer	Position in key

WARNING

The OID column is no longer used on system tables. Use UID instead.

8.4. PROCEDURE METADATA

SYS.Procedures

This table supplies information about the procedures in the virtual database.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
Name	string	Procedure name
NameInSource	string	Procedure name in source system

Column Name	Type	Description
ReturnsResults	boolean	Returns a result set
UID	string	Procedure UID
OID	integer	Unique ID
Description	string	Description

SYS.ProcedureParams

This supplies information on procedure parameters.

Column Name	Type	Description
VDBName	string	VDB name
SchemaName	string	Schema Name
ProcedureName	string	Procedure name
Name	string	Parameter name
DataType	string	Data Virtualization runtime data type name
Position	integer	Position in procedure args
Type	string	Parameter direction: "In", "Out", "InOut", "ResultSet", "ReturnValue"
Optional	boolean	Parameter is optional
Precision	integer	Precision of parameter
TypeLength	integer	Length of parameter value
Scale	integer	Scale of parameter
Radix	integer	Radix of parameter
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"

WARNING

The OID column is no longer used on system tables. Use UID instead.

8.5. DATA TYPE METADATA

SYS.DataTypes

This table supplies information on data types. See Section 4.1, “Supported Types”.

Column Name	Type	Description
Name	string	JBoss Data Virtualization design-time type name
IsStandard	boolean	Always false
IsPhysical	boolean	Always false
TypeName	string	Design-time type name (same as Name)
JavaClass	string	Java class returned for this type
Scale	integer	Max scale of this type
TypeLength	integer	Max length of this type
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
IsSigned	boolean	Is signed numeric?
IsAutoIncremented	boolean	Is auto-incremented?
IsCaseSensitive	boolean	Is case-sensitive?
Precision	integer	Max precision of this type
Radix	integer	Radix of this type
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", "Unsearchable"
UID	string	Data type unique ID

Column Name	Type	Description
OID	integer	Unique ID
RuntimeType	string	JBoss Data Virtualization runtime data type name
BaseType	string	Base type
Description	string	Description of type

WARNING

The OID column is no longer used on system tables. Use UID instead.

8.6. SYSTEM PROCEDURES

SYS.getXMLSchemas

Returns a result set with a single column, schema, containing the schemas as clobs.

```
SYS.getXMLSchemas(document in string) returns schema string
```

SYSADMIN.logMsg

Log a message to the underlying logging system.

```
SYSADMIN.logMsg(logged RETURN boolean, level IN string, context IN
string, msg IN object)
```

Returns true if the message was logged. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'

SYSADMIN.isLoggable

Tests if logging is enabled at the given level and context.

```
SYSADMIN.isLoggable(loggable RETURN boolean, level IN string, context IN
string)
```

Returns true if logging is enabled. level can be one of the log4j levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE. level defaults to 'DEBUG' and context defaults to 'org.teiid.PROCESSOR'

SYSADMIN.refreshMatView

Returns integer RowsUpdated. -1 indicates a load is in progress, otherwise the cardinality of the table is returned. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more information.

```
SYSADMIN.refreshMatView(RowsUpdated return integer, ViewName in string,
Invalidate in boolean)
```

SYSADMIN.refreshMatViewRow

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. 0 indicates that the specified row did not exist in the live data query or in the materialized table. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more information.

```
SYSADMIN.refreshMatViewRow(RowsUpdated return integer, ViewName in
string, Key in object)
```

8.7. METADATA PROCEDURES

SYSADMIN.setTableStats

Set statistics for the given table.

```
SYSADMIN.setTableStats(TableName in string, Cardinality in integer)
```

SYSADMIN.setColumnStats

Set statistics for the given column.

```
SYSADMIN.setColumnStats(TableName in string, ColumnName in string,
DistinctCount in integer, NullCount in integer, Max in string, Min in
string)
```

All stat values are nullable. Passing a null stat value will leave corresponding metadata value unchanged.

SYSADMIN.setProperty

Set an extension metadata property for the given record. Extension metadata is typically used by translators.

```
SYSADMIN.setProperty(OldValue return clob, Uid in string, Name in
string, Value in clob)
```

Setting a value to null will remove the property.

The use of this procedure will not trigger replanning of associated prepared plans.

CHAPTER 9. VIRTUAL DATABASES

9.1. VDB DEFINITION

A VDB or virtual database definition is contained in an XML file. For .vdb archive files created in the design tool, this file is embedded in the archive and most fields can be updated through tooling. The XML schema for this file can be found in the ***EAP_HOME*/docs/teiid/schema** directory.

Example 9.1. Example VDB XML

```
<vdb name="${vdb-name}" version="${vdb-version}">

    <!-- VDB properties -->
    <property name="${property-name}" value="${property-value}" />

    <!-- UDF defined in an AS module,  see Developers Guide -->
    <property name ="lib" value ="{module-name}"></property>

    <import-vdb name="..." version="..." import-data-
policies="true|false"/>

    <!-- define a model fragment for each data source -->
    <model visible="true" name="${model-name}" type="${model-type}" >

        <property name="..." value="..." />

        <source name="${source-name}" translator-name="${translator-
name}" connection-jndi-name="${deployed-jndi-name}">

        <metadata type="${repository-type}">raw text</metadata>

    </model>

   <!-- define a model with multiple sources - see Multi-Source Models -
->
   <model name="${model-name}" path="/Test/Customers.xmi">
        <property name="multisource" value="true"/>
        . . .
        <source name="${source-name}"
            translator-name="${translator-name}" connection-jndi-
name="${deployed-jndi-name}"/>
        <source . . . />
        <source . . . />
    </model>

    <!-- see Reference Guide - Data Roles -->
    <data-role name="${role-name}">
        <description>${role-description}</description>
        . . .
    </data-role>

    <!-- create translator instances that override default properties --
>
    <translator name="${translator-name}" type="${translator-type}" />
```

```
            <property name="..." value="..." />

        </translator>
    </vdb>
```

9.2. VDB DEFINITION: THE VDB ELEMENT

Attributes

- *name*

  The name of the VDB. The VDB name referenced through the driver or datasource during the connection time.

- *version*

  The version of the VDB (should be an positive integer). This determines the deployed directory location (see Name), and provides an explicit versioning mechanism to the VDB name.

Property Elements

- *cache-metadata*

  Can be "true" or "false". If "false", JBoss Data Virtualization will obtain metadata once for every launch of the VDB. "true" will save a file containing the metadata into the ***EAP_HOME*/*MODE*/data`** directory. Defaults to "false" for **`-vdb.xml`** deployments otherwise "true".

- *query-timeout*

  Sets the default query timeout in milliseconds for queries executed against this VDB. 0 indicates that the server default query timeout should be used. Defaults to 0. Will have no effect if the server default query timeout is set to a lesser value. Note that clients can still set their own timeouts that will be managed on the client side.

- *lib*

  Set to a list of modules for the VDB classpath for user defined function loading. See also Support for Non-Pushdown User Defined Functions in *Red Hat JBoss Data Virtualization Development Guide: Server Development*.

9.3. VDB DEFINITION: THE IMPORT-VDB ELEMENT

Attributes

- *name* The name of the VDB to be imported.
- *version* The version of the VDB to be imported (should be an positive integer).
- *import-data-policies* Optional attribute to indicate whether the data policies should be imported as well. Defaults to TRUE.

9.4. VDB DEFINITION: THE MODEL ELEMENT

Attributes

- *name*

  The name of the model is used as a top level schema name for all of the metadata imported from the connector. The name must be unique among all Models in the VDB and must not contain the '.' character.

- *visible*

  By default this value is set to "true". When the value is set to "false", this model will not be visible to JDBC metadata queries. Usually it is used to hide a model from client applications that must not directly issue queries against it. However, this does not prohibit either client applications or other view models from using it, if they know its schema.

Source Element

A source is a named binding of a translator and connection source to a model.

- *name*

  The name of the source to use for this model. This can be any name you like, but will typically be the same as the model name. Having a name different from the model name is only useful in multi-source scenarios. In multi-source, the source names under a given model must be unique. If you have the same source bound to multiple models it may have the same name for each. An exception will be raised if the same source name is used for different sources.

- *translator-name*

  The name or type of the JBoss Data Virtualization Translator to use. Possible values include the built-in types (ws, file, ldap, oracle, sqlserver, db2, derby, etc.) and translators defined in the translators section.

- *connection-jndi-name*

  The JNDI name of this source's connection factory. There should be a corresponding "-ds.xml" file that defines the connection factory in the JBoss EAP. Check out the deploying VDB dependencies section for info. You also need to deploy these connection factories before you can deploy the VDB.

Property Elements

- *importer.<propertyname>*

  Property to be used by the connector importer for the model for purposes importing metadata. See possible property name/values in the Translator specific section. Note that using these properties you can narrow or widen the data elements available for integration.

Metadata Element

- The optional metadata element defines the metadata repository type and optional raw metadata to be consumed by the metadata repository.

  - *type*

The metadata repository type. Defaults to INDEX for Designer VDBs and NATIVE for non-Designer VDB source models. For all other deployments/models a value must be specified. Built-in types include DDL, NATIVE, INDEX, and DDL-FILE. The usage of the raw text varies with the by type. The raw text is not used with NATIVE and INDEX (only for Designer VDBs) metadata repositories. The raw text for DDL is expected to be a series of DDL statements that define the schema. DDL-FILE (used only with zip deployments) is similar to DDL, except that the raw text specifies an absolute path relative to the vdb root of the location of a file containing the DDL. See also about a Custom Metadata Repository in *Red Hat JBoss Development Guide: Server Development*.

See Also:

- Section 12.1, “DDL Metadata”

9.5. VDB DEFINITION: THE TRANSLATOR ELEMENT

Attributes

- *name*

  The name of the Translator. Referenced by the source element.

- *type*

  The base type of the Translator. Can be one of the built-in types (ws, file, ldap, oracle, sqlserver, db2, derby, etc.).

Property Elements

- Set a value that overrides a translator default property. See possible property name/values in the Translator specific section.

9.6. DYNAMIC VDBS

Data integration is also available via a "Dynamic VDB" without the need for Teiid Designer tooling. Dynamic VDBs can be deployed either by XML or ZIP. Example files are provided with the installation of JBoss Data Virtualization.

9.7. DYNAMIC VDB XML DEPLOYMENT

You can create a ***NAME*-vdb.xml** file. The XML file captures information about the VDB, the sources it integrates, and preferences for importing metadata.

NOTE

The VDB name pattern must adhere to "-vdb.xml" for the VDB deployer to recognize this file.

The XML schema for these files is found in ***EAP_HOME*/docs/teiid/schema/vdb-deployer.xsd**.

9.8. DYNAMIC VDB ZIP DEPLOYMENT

For more complicated scenarios you can deploy a VDB via a ZIP file similar. In a VDB ZIP deployment:

- The deployment must end with the extension **`.vdb`**.
- The VDB XML file must be named **`vdb.xml`** and placed in the ZIP under the **`META-INF`** directory.
- If a **`lib`** folder exists, any JARs found underneath will automatically be added to the VDB classpath.
- For backwards compatibility with Teiid Designer VDBs, if any **`.INDEX`** file exists, the default metadata repository will be assumed to be INDEX.
- Files within the VDB ZIP are accessible by a Custom Metadata Repository using the **`MetadataFactory.getVDBResources()`** method, which returns a map of all **`VDBResources`** in the VDB keyed by absolute path relative to the VDB root. See *Red Hat JBoss Data Virtualization Development Guide: Server Development* for more information about custom metadata repositories.
- The built-in **`DDL-FILE`** metadata repository type may be used to define DDL-based metadata in files outside of the **`vdb.xml`**. This improves the memory footprint of the VDB metadata and the maintainability of **`vdb.xml`**.

Example 9.2. Example VDB Zip Structure

```
/META-INF
   vdb.xml
/ddl
   schema1.ddl
/lib
   some-udf.jar
```

In the above example the **`vdb.xml`** could use a **`DDL-FILE`** metadata type for **`schema1`**:

```
<model name="schema1" ...
   <metadata type="DDL-FILE">/ddl/schema1.ddl<metadata>
</model>
```

9.9. VDB REUSE

VDBs may reuse other VDBs deployed in the same server instance by using an "import-vdb" declaration in the vdb.xml file (see Section 9.1, “VDB Definition”). An imported VDB can have its tables and procedures referenced by views and procedures in the importing VDB as if they are part of the VDB. Imported VDBs are required to exist before an importing VDB may start. If an imported VDB is undeployed, then any importing VDB will be stopped.

An imported VDB includes all of its models and may not conflict with any model, data policy, or source already defined in the importing VDB. Once a VDB is imported it is mostly operationally independent from the base VDB. Only cost related metadata may be updated for an object from an imported VDB in the scope of the importing VDB. All other updates must be made through the original VDB, but they will be visible in all imported VDBs. Even materialized views are separately maintained for an imported VDB in the scope of each importing VDB.

■

Example 9.3. Example reuse VDB XML

```
<vdb name="reuse" version="1">

    <import-vdb name="common" version="1" import-data-policies="false"/>

    <model visible="true" type="VIRTUAL" name="new-model">
         <metadata type = "DDL"><![CDATA[
              CREATE VIEW x (
                y varchar
                ) AS
                  select * from old-model.tbl;
         ]]>
         </metadata>
    </model>
</vdb>
```

In the above example the reuse VDB will have access to all of the models defined in the common VDB and adds in the "new-model".

9.10. METADATA REPOSITORIES

Traditionally the metadata for a Virtual Database is built by Teiid Designer and supplied to Teiid engine through a VDB archive file. This VDB file contains .INDEX metadata files. By default they are loaded by a MetadataRepository with the name INDEX. Other built-in metadata repositories include the following:

NATIVE

This is only applicable on source models (and is also the default), when used the metadata for the model is retrieved from the source database itself.

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE"></metadata>
    </model>
</vdb>
```

DDL

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="DDL">
          **DDL Here**
        </metadata>
    </model>
</vdb>
```

This is applicable to both source and view models. See DDL Metadata for more information on how to use this feature.

DDL-FILE

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="DDL-FILE">/accounts.ddl</metadata>
    </model>
</vdb>
```

DDL is applicable to both source and view models in zip VDB deployments. See DDL Metadata for more information on how to use this feature.

Chaining Repositories

When defining the metadata type for a model, multiple metadata elements can be used. All the repository instances defined are consulted in the order configured to gather the metadata for the given model.

```
<vdb name="{vdb-name}" version="1">
    <model name="{model-name}" type="PHYSICAL">
        <source name="AccountsDB" translator-name="oracle" connection-
jndi-name="java:/oracleDS"/>
        <metadata type="NATIVE"/>
        <metadata type="DDL">
          **DDL Here**
        </metadata>
    </model>
</vdb>
```

For the above model, NATIVE importer is first used, then DDL importer used to add additional metadata to NATIVE imported metadata.

CHAPTER 10. GENERATED REST SERVICES

10.1. GENERATED REST SERVICES

Using DDL metadata, properties can be specified that enable JBoss Data Virtualization procedures to be exposed as a REST based services.

When a VDB includes this metadata and is deployed in JBoss EAP, and if the VDB is valid and after the metadata is loaded, then a REST war is generated automatically and deployed into the local JBoss EAP server.

10.2. REST PROPERTIES

The following properties can be specified on a JBoss Data Virtualization virtual procedure.

Property Name	Description	Is Required	Allowed Values
METHOD	HTTP Method to use	Yes	GET | POST| PUT | DELETE
URI	URI of procedure	Yes	ex:/procedure
PRODUCES	Type of content produced by the service	no	xml | json | plain | any text
CHARSET	When procedure returns Blob, and content type text based, this character set to used to convert the data	no	US-ASCII | UTF-8

The above properties must be defined with NAMESPACE 'http://teiid.org/rest' on the metadata. Here is an example VDB that defines the REST based service.

10.3. EXAMPLE VDB WITH REST PROPERTIES

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="sample" version="1">
    <property name="UseConnectorMetadata" value="true" />
    <property name="{http://teiid.org/rest}auto-generate" value="true"/>

    <model name="PM1">
        <source name="text-connector" translator-name="loopback" />
         <metadata type="DDL"><![CDATA[
                CREATE FOREIGN TABLE G1 (e1 string, e2 integer);
                CREATE FOREIGN TABLE G2 (e1 string, e2 integer);
        ]]> </metadata>
    </model>
    <model name="View" type ="VIRTUAL">
         <metadata type="DDL"><![CDATA[
            SET NAMESPACE 'http://teiid.org/rest' AS REST;
            CREATE VIRTUAL PROCEDURE g1Table(IN p1 integer) RETURNS TABLE
(xml_out xml) OPTIONS (UPDATECOUNT 0, "REST:METHOD" 'GET', "REST:URI"
'g1/{p1}')
            AS
            BEGIN
                SELECT XMLELEMENT(NAME "rows", XMLATTRIBUTES (g1Table.p1
as p1), XMLAGG(XMLELEMENT(NAME "row", XMLFOREST(e1, e2)))) AS xml_out FROM
PM1.G1;
            END
            ]]> </metadata>
    </model>

</vdb>
```

The REST VDB is deployed with "{vdb-name}_{vdb-version}" context. The model name is prepended to uri of the service call. For example the procedure in above example can be accessed as

```
http://{host}:8080/sample_1/view/g1/123
```

where "sample_1" is context, "view" is model name, "g1" is URI, and 123 is parameter {p1} from URI.

NOTE

<property name="{ http://teiid.org/rest }auto-generate" value="true"/>, can be used to control the generation of the REST based WAR based on the VDB. This property along with at least one procedure with REST based extension metadata is required to generate a REST WAR file. Also, the procedure will return the result set with single column of either XML, CLOB, BLOB or String. When PRODUCES property is not defined, this property is derived from the result column that is projected out.

When designing the procedures that will be invoked through GET based call, the input parameters for procedures can be defined in the PATH of the URI, as the {p1} example above, or they can also be defined as query parameter, or combination of both. Here is an example:

```
http://{host}:8080/sample_1/view/g1?p1=123
http://{host}:8080/sample_1/view/g1/123?p2=foo
```

Make sure that the number of parameters defined on the URI and query match to the parameters defined on procedure definition. If you defined a default value for a parameter on the procedure, and that parameter going to be passed in query parameter on URL then you have choice to omit that query parameter, if you defined as PATH you must supply a value for it.

'POST' methods MUST not be defined with URI with PATHS for parameters as in GET operations, the procedure parameters are automatically added as @FormParam annotations on the generated procedure. A client invoking this service must use FORM to post the values for the parameters. The FORM field names MUST match the names of the procedure parameters names.

If any one of the procedure parameters are BLOB, CLOB or XML type, then POST operation can be only invoked using "multipart/form-data" RFC-2388 protocol. This allows user to upload large binary or XML files efficiently to Teiid using streaming".

If a parameter to the procedure is VARBINARY type then the value of the parameter must be properly BASE64 encoded, irrespective of the HTTP method used to execute the procedure. If this VARBINARY has large content, then consider using BLOB.

10.4. CONSIDERATIONS FOR GENERATED REST SERVICES

If you defined a procedure that returns a XML content, then REST service call must be called with "accepts" HTTP header of "application/xml". Also, if you defined a procedure that returns a JSON content and PRODUCES property is defined "json" then HTTP client call must include the "accepts" header of "application/json". In the situations where "accepts" header is missing, and only one procedure is defined with unique path, that procedure will be invoked. If there are multiple procedures with same URI path, for example one generating XML and another generating JSON content, then "accepts" header directs the REST engine as to which procedure will be invoked to get the results. A wrong "accepts" header will result in error.

WARNING

Ensure the number of parameters defined on the URI must match to the parameters defined on procedure definition. An error with parameter definition will result in procedure being skipped from generation of REST based service or error with 'GET' based methods. 'POST' methods do not need to be defined with URI paths, the procedure parameters are automatically added as @FormParam annotations on the generated procedure.

10.5. SECURITY FOR GENERATED REST SERVICES

By default all the generated Rest based services are secured using "HTTPBasic" with security domain "teiid-security" and with security role "rest". However, these properties can be customized by defining the then in vdb.xml file.

Example 10.1. Example vdb.xml file security specification

```
<vdb name="sample" version="1">
    <property name="UseConnectorMetadata" value="true" />
    <property name="{http://teiid.org/rest}auto-generate" value="true"/>
    <property name="{http://teiid.org/rest}security-type"
value="HttpBasic"/>
    <property name="{http://teiid.org/rest}security-domain"
value="teiid-security"/>
    <property name="{http://teiid.org/rest}security-role"
value="example-role"/>
    <property name="{http://teiid.org/rest}passthrough-auth"
value="true"/>

    ...
</vdb>
```

- security-type - defines the security type. allowed values are "HttpBasic" or "none". If omitted will default to "HttpBasic"
- security-domain - defines JAAS security domain to be used with HttpBasic. If omitted will default to "teiid-security"

- security-role - security role that HttpBasic will use to authorize the users. If omitted the value will default to "rest"
- passthough-auth - when defined the pass-through-authentication is used to login in to JBoss Data Virtualization. When this is set to "true", make sure that the "embedded" transport configuration in **`standalone.xml`** has defined a security-domain that can be authenticated against. Failure to add the configuration change will result in authentication error. Defaults to false.

IMPORTANT

it is our intention to provide other types of security based on ws-security in future releases.

10.6. AD-HOC REST SERVICES

Apart from the explicitly defined procedure based rest services, the generated jax-rs war file will also implicitly include a special rest based service under URI "/query" that can take any XML or JSON producing SQL as parameter and expose the results of that query as result of the service. This service is defined with "POST", accepting a Form Parameter named "sql". For example, after you deploy the VDB defined in above example, you can issue a HTTP POST call as

```
    http://localhost:8080/sample_1/view/query
    sql=SELECT XMLELEMENT(NAME "rows",XMLAGG(XMLELEMENT(NAME "row",
XMLFOREST(e1, e2)))) AS xml_out FROM PM1.G1
```

A sample HTTP Request from Java can be made like below:

```

        public static String httpCall(String url, String method, String
params) throws Exception {
                StringBuffer buff = new StringBuffer();
                HttpURLConnection connection = (HttpURLConnection) new
URL(url).openConnection();
                connection.setRequestMethod(method);
                connection.setDoOutput(true);

                if (method.equalsIgnoreCase("post")) {
                        OutputStreamWriter wr = new
OutputStreamWriter(connection.getOutputStream());
                    wr.write(params);
                    wr.flush();
                }

                BufferedReader serverResponse = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                String line;
                while ((line = serverResponse.readLine()) != null) {
                        buff.append(line);
                }
                return buff.toString();
        }

        public static void main(String[] args) throws Exception {
            String params = URLEncoder.encode("sql", "UTF-8") + "=" +
URLEncoder.encode("SELECT XMLELEMENT(NAME "rows",XMLAGG(XMLELEMENT(NAME
"row", XMLFOREST(e1, e2)))) AS xml_out FROM PM1.G1", "UTF-8");
            httpCall("http://localhost:8080/sample_1/view/query", "POST",
params);
        }
```

CHAPTER 11. MULTI-SOURCE MODELS

11.1. MULTI-SOURCE MODELS

Multi-source models can be used to quickly access data in multiple sources with homogeneous metadata. When you have multiple instances of data that are using identical schema (horizontal sharding), JBoss Data Virtualization can help you aggregate data across all the instances, using "multi-source" models. In this scenario, instead of creating/importing a model for every data source, user must define one source model that represents the schema and configure multiple data "sources" underneath it. During runtime, when a query issued against this model, the query engine analyzes the information and gathers the required data from all the sources configured and aggregates the results and provides in a single result set. Since all sources use the same physical metadata, this feature is most appropriate for accessing the same source type with multiple instances.

11.2. MULTI-SOURCE MODEL CONFIGURATION

To mark a model as multi-source, multisource can be set to **`true`** and then more than one source can be listed for the model in the **`vdb.xml`** file. The following example shows a single model dynamic VDB with multiple sources defined.

```
<vdb name="vdbname" version="1">
    <model visible="true" type="PHYSICAL" name="Customers"
path="/Test/Customers.xmi">
        <property name="multisource" value="true"/>
        <!-- optional properties
        <property name="multisource.columnName" value="somename"/>
        <property name="multisource.addColumn" value="true"/>
        -->
        <source name="chicago"
            translator-name="oracle" connection-jndi-name="chicago-
customers"/>
        <source name="newyork"
            translator-name="oracle" connection-jndi-name="newyork-
customers"/>
        <source name="la"
            translator-name="oracle" connection-jndi-name="la-
customers"/>
    </model>
</vdb>
```

NOTE

Currently the tooling support for managing the multi-source feature is limited, so if you need to use this feature build the VDB as usual in the Teiid Designer and then edit the **`vdb.xml`** file in the VDB archive using a Text editor to add the additional sources as defined above. You must deploy a separate data source for each source defined in the XML file.

In the above example, the VDB defined has single model called **`Customers`**, that has multiple sources (**`chicago`**, **`newyork`**, and **`la`**) that define different instances of data.

11.3. THE MULTI-SOURCE COLUMN

When a model is marked as multi-source, the engine will add or use an existing column on each table to represent the source name values. In the above vdb.xml the column would return **`chicago`** , **`la`** , **`newyork`** for each of the respective sources. The name of the column defaults to SOURCE_NAME, but is configurable by setting the model property *multisource.columnName* . If a column already exists on the table (or an IN procedure parameter) with the same name, the engine will assume that it should represent the multi-source column and it will not be used to retrieve physical data. If the multi-source column is not present, the generated column will be treated as a pseudo column which is not selectable via wildcards (* nor tbl.*).

This allows queries like the following:

```
select * from table where SOURCE_NAME = 'newyork'
update table column=value where SOURCE_NAME='chicago'
delete from table where column = x and SOURCE_NAME='la'
insert into table (column, SOURCE_NAME) VALUES ('value', 'newyork')
```

11.4. THE MULTI-SOURCE COLUMN IN SYSTEM METADATA

The pseudo column is by default not present in your actual metadata; it is not added on source tables/procedures when you import the metadata. If you would like to use the multi-source column in your transformations to control which sources are accessed or updated or you would like the column reported via metadata facilities, there are several options:

- With either VDB type to make the multi-source column present in the system metadata, you can set the model property *multisource.addColumn* to true on a multi-source model. Care must be taken though when using this property in Teiid Designer as any transformation logic (views/procedures) that you have defined will not have been aware of the multi-source column and may fail validation upon server deployment.

- If using Teiid Designer, you can manually add the multi-source column.

- If using Dynamic VDBs, the pseudo-column will already be available to transformations, but will not be present in your System metadata by default. If you are using DDL and you would like to be selective (rather than using the *multisource.addColumn* property), you can manually add the column via DDL.

11.5. MULTI-SOURCE MODELS: PLANNING AND EXECUTION

The planner logically treats a multi-source table as if it were a view containing the union all of the respective source tables. More complex partitioning scenarios, such as heterogeneous sources or list partitioning will require the use of a Partitioned Union.

Most of the federated optimizations available over unions are still applicable in multi-source mode. This includes aggregation pushdown/decomposition, limit pushdown, join partitioning, etc.

11.6. MULTI-SOURCE MODELS: SELECT, UPDATE AND DELETE

- A multi-source query against a SELECT/UPDATE/DELETE may affect any subset of the sources based upon the evaluation of the WHERE clause.

- The multi-source column may not be targeted in an update change set.

- The sum of the update counts for UPDATEs/DELETEs will be returned as the resultant update count.

- When running under a transaction in a mode that detects the need for a transaction and multiple updates may performed or a transactional read is required and multiple sources may be read from, a transaction will be started to enlist each source.

11.7. MULTI-SOURCE MODELS: INSERT

- A multi-source INSERT must use the source_name column as an insert column to specify which source will be targeted by the INSERT. Only an INSERT using the VALUES clause is supported.

11.8. MULTI-SOURCE MODELS: STORED PROCEDURES

A physical stored procedure requires the addition of a string in parameter matching the multi-source column name to specify which source the procedure is executed on. If the parameter is not present and defaults to a null value, then the procedure will be executed on each source. It is not possible to execute procedures that are required to return IN/OUT, OUT, or RETURN parameters values on more than 1 source.

Example 11.1. Example DDL

```
CREATE FOREIGN PROCEDURE PROC (arg1 IN STRING NOT NULL, arg2 IN STRING,
SOURCE_NAME IN STRING)
```

Example 11.2. Example Calls Against A Single Source

```
CALL PROC(arg1=>'x', SOURCE_NAME=>'sourceA')
EXEC PROC('x', 'y', 'sourceB')
```

Example 11.3. Example Calls Against All Sources

```
CALL PROC(arg1=>'x')
EXEC PROC('x', 'y')
```

CHAPTER 12. DDL METADATA

12.1. DDL METADATA

A VDB can define models/schemas using DDL. Here is a small example of how one can define a view inside the **-vdb.xml** file. See the **<metadata>** element under **<model>**.

Example 12.1. Example to show view definition

```
<model visible = "true" type = "VIRTUAL" name = "customers">
     <metadata type = "DDL"><![CDATA[
              CREATE VIEW PARTS (
                   PART_ID integer PRIMARY KEY,
                   PART_NAME varchar(255),
                   PART_COLOR varchar(30),
                   PART_WEIGHT varchar(255)
               ) AS
                  select a.id as PART_ID, a.name as PART_NAME, b.color
as PART_COLOR, b.weight as PART_WEIGHT from modelA.part a, modelB.part b
where a.id = b.id
     ]]>
     </metadata>
</model>
```

Another complete DDL based example is at the end of this section.

NOTE

The declaration of metadata using DDL, **NATIVE** or **DDL-FILE** is supported out of the box, however the MetadataRepository interface allows users to plug-in their own metadata facilities. For example, you can write a Hibernate based store that can feed the necessary metadata. You can find out more about custom metadata repositories in *Red Hat JBoss Data Virtualization Development Guide: Server Development*.

NOTE

The DDL based schema is not constrained to be defined only for the view models.

NOTE

The full grammar for DDL is in the appendix.

12.2. FOREIGN TABLE

A FOREIGN table is a table that is defined on a physical model that represents a real relational table in source databases like Oracle, SQLServer etc. For relational databases, JBoss Data Virtualization has the capability to automatically retrieve the database schema information upon the deployment of the VDB, if one like to auto import the existing schema. However, the user can use below FOREIGN table semantics, when they would like to explicitly define tables on PHYSICAL models or represent non-relational data as relational data in custom translators.

Example 12.2. Example:Create Foreign Table(Created on PHYSICAL model)

```
CREATE FOREIGN TABLE Customer (id integer PRIMARY KEY, firstname
varchar(25), lastname varchar(25),  dob timestamp);

CREATE FOREIGN TABLE Order (id integer PRIMARY KEY, customerid integer,
saledate date, amount decimal(25,4), CONSTRAINT fk FOREGIN
KEY(customerid) REFERENCES Customer(id));
```

NOTE

See "create table" in Section A.7, “Productions”.

12.3. VIEW

A view is a virtual table. A view contains rows and columns,like a real table. The fields in a view are fields from one or more real tables from the source or other view models. They can also be expressions made up multiple columns, or aggregated columns. When column definitions are not defined on the view table, they will be derived from the projected columns of the view's select transformation that is defined after the AS keyword.

You can add functions, JOIN statements and WHERE clauses to a view data as if the data were coming from one single table.

NOTE

See "create table" in Section A.7, “Productions”.

12.4. TABLE OPTIONS

You can use the following options when creating a table or view. See "create table body" in Section A.7, “Productions”. Any others properties defined will be considered as extension metadata.

Property	Data Type or Allowed Values	Description
UUID	string	Unique identifier for View
MATERIALIZED	'TRUE'|'FALSE'	Defines if a table is materialized
MATERIALIZED_TABLE	'table.name'	If this view is being materialized to a external database, this defines the name of the table that is being materialized to
CARDINALITY	int	Costing information. Number of rows in the table. Used for planning purposes
UPDATABLE	'TRUE'|'FALSE'	Defines if the view is allowed to update or not

Property	Data Type or Allowed Values	Description
ANNOTATION	string	Description of the view

Example 12.3. Example:Create View Table(Created on VIRTUAL model)

```
CREATE VIEW CustomerOrders (name varchar(50),  saledate date, amount
decimal) OPTIONS (CARDINALITY 100, ANNOTATION 'Example')
  AS
  SELECT concat(c.firstname, c.lastname) as name, o.saledate as
saledate, o.amount as amount FROM Customer C JOIN Order o ON c.id =
o.customerid;
```

12.5. COLUMN OPTIONS

You can use the following options when specifying columns in the creation of a table or view. Any others properties defined will be considered as extension metadata.

Property	Data Type or Allowed Values	Description
UUID	string	A unique identifier for the column
NAMEINSOURCE	string	If this is a column name on the FOREIGN table, this value represents name of the column in source database, if omitted the column name is used when querying for data against the source
CASE_SENSITIVE	'TRUE'|'FALSE'	
SELECTABLE	'TRUE'|'FALSE'	TRUE when this column is available for selection from the user query
UPDATABLE	'TRUE'|'FALSE'	Defines if the column is updatable. Defaults to true if the view/table is updatable.
SIGNED	'TRUE'|'FALSE'	
CURRENCY	'TRUE'|'FALSE'	
FIXED_LENGTH	'TRUE'|'FALSE'	
SEARCHABLE	'SEARCHABLE'|'UNSEARCHABLE'|'LIKE_ONLY'|'ALL_EXCEPT_LIKE'	column searchability, usually dictated by the data type

Property	Data Type or Allowed Values	Description
MIN_VALUE		
MAX_VALUE		
CHAR_OCTET_LENGTH	integer	
ANNOTATION	string	
NATIVE_TYPE	string	
RADIX	integer	
NULL_VALUE_COUNT	long	costing information. Number of NULLS in this column
DISTINCT_VALUES	long	costing information. Number of distinct values in this column

12.6. TABLE CONSTRAINTS

Constraints can be defined on table/view to define indexes and relationships to other tables/views. See "create table body" in Section A.7, “Productions”.

This information is used by the JBoss Data Virtualization optimizer to plan queries or use the indexes in materialization tables to optimize the access to the data.

CONSTRAINTS are same as one can define on RDBMS.

Example 12.4. Example of CONSTRAINTs

```
CREATE VIEW CustomerOrders (name varchar(50),  saledate date, amount
decimal,
   CONSTRAINT EXAMPLE_INDEX INDEX (name, amount)
   ACCESSPATTERN (name)
   PRIMARY KEY ...
```

12.7. INSTEAD OF TRIGGERS

A view comprising multiple base tables must use an INSTEAD OF trigger to support inserts, updates and deletes that reference data in the tables. See "create trigger" in Section A.7, “Productions”.

Based on the select transformation's complexity some times INSTEAD OF Triggers are automatically provided for the user when "UPDATABLE" OPTION on the view is set to "TRUE". However, using the CREATE TRIGGER mechanism user can provide/override the default behaviour.

Example 12.5. Example:Define instead of trigger on View

```
CREATE TRIGGER ON CustomerOrders INSTEAD OF INSERT AS
   FOR EACH ROW
   BEGIN ATOMIC
   INSERT INTO Customer (...) VALUES (NEW.value ...);
   END
```

12.8. PROCEDURES AND FUNCTIONS

A user can define one of the following functions:

- Source Procedure ("CREATE FOREIGN PROCEDURE") - a stored procedure in source
- Source Function ("CREATE FOREIGN FUNCTION") - A function that is supported by the source, where JBoss Data Virtualization will pushdown to source instead of evaluating in the JBoss Data Virtualization engine.
- Virtual Procedure ("CREATE VIRTUAL PROCEDURE") - Similar to stored procedure, however this is defined using the JBoss Data Virtualization Procedure language and evaluated in the JBoss Data Virtualization engine.
- Function/UDF ("CREATE VIRTUAL FUNCTION") - A user defined function, that can be defined using the Teiid procedure language or can have the implementation defined using a JAVA Class.

See "create procedure" in Section A.7, “Productions”.

12.9. VARIABLE ARGUMENT SUPPORT

Instead of using just an IN parameter, the last non optional parameter can be declared VARIADIC to indicate that it can be repeated 0 or more times when the procedure is called positionally. Section A.7, “Productions”.

Example 12.6. Example:Vararg procedure

```
CREATE FOREIGN PROCEDURE proc (x integer, VARIADIC z integer) returns (x
string);
```

12.10. FUNCTION OPTIONS

You can use the following options when creating functions. See "create procedure" in Section A.7, “Productions”. Any others properties defined will be considered as extension metadata.

Property	Data Type or Allowed Values	Description
UUID	string	unique Identifier

Property	Data Type or Allowed Values	Description
NAMEINSOURCE	If this is source function/procedure the name in the physical source, if different from the logical name given above	
ANNOTATION	string	Description of the function/procedure
CATEGORY	string	Function Category
DETERMINISM	● NONDETERMINISTIC ● COMMAND_DETERMINISTIC ● SESSION_DETERMINISTIC ● USER_DETERMINISTIC ● VDB_DETERMINISTIC ● DETERMINISTIC	
NULL-ON-NULL	'TRUE'|'FALSE'	
JAVA_CLASS	string	Java Class that defines the method in case of UDF
JAVA_METHOD	string	The Java method name on the above defined java class for the UDF implementation
VARARGS	'TRUE'|'FALSE'	Indicates that the last argument of the function can be repeated 0 to any number of times. default false. It is more proper to use a VARIADIC parameter.
AGGREGATE	'TRUE'|'FALSE'	Indicates the function is a user defined aggregate function. Properties specific to aggregates are listed below:

Note that NULL-ON-NULL, VARARGS, and all of the AGGREGATE properties are also valid relational extension metadata properties that can be used on source procedures marked as functions.

You can also create FOREIGN functions that are supported by a source. See the section on user defined functions in *Red Hat JBoss Data Virtualization Development Guide: Server Development*for more information about source supported functions.

12.11. AGGREGATE FUNCTION OPTIONS

Property	Data Type or Allowed Values	Description
ANALYTIC	'TRUE'|'FALSE'	indicates the aggregate function must be windowed. default false.
ALLOWS-ORDERBY	'TRUE'|'FALSE'	indicates the aggregate function supports an ORDER BY clause. default false
ALLOWS-DISTINCT	'TRUE'|'FALSE'	indicates the aggregate function supports the DISTINCT keyword. default false
DECOMPOSABLE	'TRUE'|'FALSE'	indicates the single argument aggregate function can be decomposed as agg(agg(x)) over subsets of data. default false
USES-DISTINCT-ROWS	'TRUE'|'FALSE'	indicates the aggregate function effectively uses distinct rows rather than all rows. default false

Note that virtual functions defined using the Teiid procedure language cannot be aggregate functions.

NOTE

If you have defined a UDF (virtual) function without a Teiid procedure definition, then it must be accompanied by its implementation in Java. To configure the Java library as dependency to the VDB, see Support for User-Defined Functions in *Red Hat JBoss Data Virtualization Development Guide: Server Development*.

12.12. PROCEDURE OPTIONS

You can use the following options when creating procedures. See "create procedure" in Section A.7, “Productions”. Any others properties defined will be considered as extension metadata.

Property	Data Type or Allowed Values	Description
UUID	string	Unique Identifier
NAMEINSOURCE	string	In the case of source
ANNOTATION	string	Description of the procedure
UPDATECOUNT	int	if this procedure updates the underlying sources, what is the update count, when update count is >1 the XA protocol for execution is enforced

Example 12.7. Example:Define Procedure

```
CREATE VIRTUAL PROCEDURE CustomerActivity(customerid integer) RETURNS
(name varchar(25), activitydate date, amount decimal) AS
    BEGIN
   ...
   END
```

Example:Define Virtual Function

```
CREATE VIRTUAL FUNCTION CustomerRank(customerid integer) RETURNS integer
AS
   BEGIN
   ...
   END
```

12.13. OPTIONS

Options can be provided for several commands. See "options clause" in Section A.7, “Productions”.

NOTE

Any option name of the form prefix:key will attempt to be resolved against the current set of namespaces. Failure to resolve will result in the option name being left as is. A resolved name will be replaced with {uri}key. See also Namespaces for Extension Metadata.

Options can also be added using the ALTER statement.

12.14. ALTER STATEMENT

ALTER statements currently primarily support adding OPTIONS properties to Tables, Views and Procedures. Using a ALTER statement, you can either add, modify or remove a property.

See "alter column options", "alter options", and "alter options list" in Section A.7, “Productions”.

Example 12.8. Example ALTER

```
ALTER FOREIGN TABLE "customer" OPTIONS (ADD CARDINALITY 10000);
ALTER FOREIGN TABLE "customer" ALTER COLUMN "name" OPTIONS(SET UPDATABLE
FALSE)
```

ALTER statements are especially useful, when user would like to modify/enhance the metadata that has been imported from a NATIVE datasource. For example, if you have a database called "northwind", and you imported that metadata and would like to add CARDINALITY to its "customer" table, you can use ALTER statement, along with "chainable" metadata repositories feature to add this property to the desired table. The below shows an example -vdb.xml file, that illustrates the usage.

Example 12.9. Example VDB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="northwind" version="1">
    <model name="nw">
        <property name="importer.importKeys" value="true"/>
        <property name="importer.importProcedures" value="true"/>
         <source name="northwind-connector" translator-name="mysql"
connection-jndi-name="java:/nw-ds"/>
         <metadata type = "NATIVE,DDL"><![CDATA[
              ALTER FOREIGN TABLE "customer" OPTIONS (ADD CARDINALITY
10000);
              ALTER FOREIGN TABLE "customer" ALTER COLUMN "name"
OPTIONS(SET UPDATABLE FALSE);
         ]]>
         </metadata>
    </model>
</vdb>
```

12.15. NAMESPACES FOR EXTENSION METADATA

When defining the extension metadata in the case of Custom Translators, the properties on tables/views/procedures/columns can define namespace for the properties such that they will not collide with properties specific to JBoss Data Virtualization. The property should be prefixed with alias of the Namespace. Prefixes starting with teiid_ are reserved for use by JBoss Data Virtualization.

See "option namespace" in Section A.7, “Productions”.

Example 12.10. Example of Namespace

```
SET NAMESPACE 'http://custom.uri' AS foo

CREATE VIEW MyView (...) OPTIONS ("foo:mycustom-prop" 'anyvalue')
```

Table 12.1. Built-in Namespace Prefixes

Prefix	URI	Description
teiid_rel	http://www.teiid.org/ext/relational/2012	Relational extensions. Uses include function and native query metadata
teiid_sf	http://www.teiid.org/translator/salesforce/2012	Salesforce extensions.

12.16. EXAMPLE DDL METADATA

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="twitter" version="1">

    <description>Shows how to call Web Services</description>
    <property name="UseConnectorMetadata" value="cached" />

    <model name="twitter">
        <source name="twitter" translator-name="rest" connection-jndi-
name="java:/twitterDS"/>
    </model>
    <model name="twitterview" type="VIRTUAL">
         <metadata type="DDL"><![CDATA[
             CREATE VIRTUAL PROCEDURE getTweets(query varchar) RETURNS
(created_on varchar(25), from_user varchar(25), to_user varchar(25),
                 profile_image_url varchar(25), source varchar(25), text
varchar(140)) AS
                select tweet.* from
                        (call twitter.invokeHTTP(action => 'GET',
endpoint =>querystring('',query as "q"))) w,
                        XMLTABLE('results' passing JSONTOXML('myxml',
w.result) columns
                        created_on string PATH 'created_at',
                        from_user string PATH 'from_user',
                        to_user string PATH 'to_user',
                        profile_image_url string PATH
'profile_image_url',
                        source string PATH 'source',
                        text string PATH 'text') tweet;
                CREATE VIEW Tweet AS select * FROM twitterview.getTweets;
        ]]> </metadata>
    </model>

    <translator name="rest" type="ws">
                <property name="DefaultBinding" value="HTTP"/>
                <property name="DefaultServiceMode" value="MESSAGE"/>
    </translator>
</vdb>
```

CHAPTER 13. TRANSLATORS

13.1. JBOSS DATA VIRTUALIZATION CONNECTOR ARCHITECTURE

The process of integrating data from an enterprise information system into JBoss Data Virtualization requires one to two components:

1. a translator (mandatory) and
2. a resource adapter (optional), also known as a connector. Most of the time, this will be a Java EE Connector Architecture (JCA) Adapter.

A translator is used to:

- translate JBoss Data Virtualization commands into commands understood by the datasource for which the translator is being used,
- execute those commands,
- return batches of results from the datasource, translated into the formats that JBoss Data Virtualization is expecting.

A resource adapter (or connector):

- handles all communications with individual enterprise information systems, (which can include databases, data feeds, flat files and so forth),
- can be a JCA Adapter or any other custom connection provider (the JCA specification ensures the writing, packaging and configuration are undertaken in a consistent manner),

NOTE

Many software vendors provide JCA Adapters to access different systems. Red Hat recommends using vendor-supplied JCA Adapters when using JMS with JCA. See http://docs.oracle.com/cd/E21764_01/integration.1111/e10231/adptr_jms.htm

- removes concerns such as connection information, resource pooling, and authentication for translators.

With a suitable translator (and optional resource adapter), any datasource or Enterprise Information System can be integrated with JBoss Data Virtualization.

13.2. TRANSLATORS

A translator acts as the bridge between JBoss Data Virtualization and an external system, which is most commonly accessed through a JCA resource adapter. Translators indicate what SQL constructs are supported and what import metadata can be read from particular datasources.

A translator is typically paired with a particular JCA resource adapter. A JCA resource adapter is not needed in instances where features such as pooling, environment dependent configuration management, or advanced security handling are not needed.

NOTE

See *Red Hat JBoss Data Virtualization Development Guide: Server Development* for more information on developing custom translators and JCA resource adapters.

See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* and the examples in ***EAP_HOME*/docs/teiid/datasources** for more information about configuring resource adapters.

13.3. TRANSLATOR PROPERTIES

Translators can have a number of configurable properties. These are divided among the following categories:

- Execution Properties - these properties determine aspects of how data is retrieved. A list of properties common to all translators are provided in Section 13.5, "Base Execution Properties".

  NOTE

  The execution properties for a translator typically have reasonable defaults. For specific translator types, base execution properties are already tuned to match the source. In most cases the user will not need to adjust their values.

- Importer Properties - these properties determine what metadata is read for import. There are no common importer properties.

NOTE

The import capabilities of translators is currently only used by dynamic VDBs and not by Teiid Designer. See Section 9.6, "Dynamic VDBs".

13.4. TRANSLATORS IN JBOSS DATA VIRTUALIZATION

JBoss Data Virtualization provides the following translators:

Apache Cassandra (Technical Preview Only)

WARNING

Technology Preview features are not supported, may not be functionally complete, and are not intended for production use. These features are included to provide customers with early access to upcoming product innovations, enabling them to test functionality and provide feedback during the development process.

Support of Apache Cassandra brings support for the popular columnar NoSQL database to JDV customers.

Apache Solr

With Apache Solr, JDV customers will be able to take advantage of enterprise search capabilities for organized retrieval of structured and unstructured data.

Cloudera Impala

Cloudera Impala support provides for fast SQL query access to data stored in Hadoop.

JDBC Translator

The JDBC Translator works with many relational databases.

JBoss Enterprise Data Services Platform Supported Configurations

File Translator

The File Translator provides a procedural way to access the file system in order to handle text files.

Google Spreadsheet Translator

The Google Spreadsheet Translator is used to connect to a Google Spreadsheet.

JBoss Data Grid 6.3

You can perform reads and writes to JDG. You can use it as an embedded cache or a remote cache.

LDAP Translator

The LDAP Translator provides access to LDAP directory services.

MongoDB Translator

The MongoDB translator, known by the type name *mongodb*, provides a relational view of data that resides in a MongoDB database. This translator is capable of converting JBoss Data Virtualization SQL queries into MongoDB based queries. It supports a full range of SELECT, INSERT, UPDATE and DELETE calls.

Object Translator

The Object translator is a bridge for reading Java objects from external sources such as JBoss Data Grid (`infinispan-cache`) or Map Cache and delivering them to the engine for processing.

OData Translator

The OData translator exposes the OData V2 and V3 data sources and uses the JBoss Data Virtualization WS resource adapter for making web service calls. This translator is an extension of the WS Translator.

OLAP Translator

The OLAP Services translator exposes stored procedures for calling analysis services backed by an OLAP server using MDX query language.

Salesforce Translator

The Salesforce Translator works with Salesforce interfaces.

Web Services Translator

The Web Services Translator provides procedural access to XML content by using *web services*.

If these translators are not suitable for your system then you can develop a custom one.

13.5. BASE EXECUTION PROPERTIES

The following execution properties are shared by all translators.

Table 13.1. Base Execution Properties

Name	Description	Default
Immutable	Set to true to indicate that the source never changes.	false
RequiresCriteria	Set to true to indicate that source SELECT/UPDATE/DELETE queries require a WHERE clause.	false
SupportsOrderBy	Set to true to indicate that the ORDER BY clause is supported.	true
SupportsOuterJoins	Set to true to indicate that OUTER JOINs are supported.	true
SupportsFullOuterJoins	If outer joins are supported, true indicates that FULL OUTER JOINs are supported.	true
SupportsInnerJoins	Set to true to indicate that INNER JOINs are supported.	true
SupportedJoinCriteria	If joins are supported, defines what criteria may be used as the join criteria. May be one of (ANY, THETA, EQUI, or KEY).	ANY
MaxInCriteriaSize	If in criteria are supported, defines what the maximum number of in entries are per predicate. -1 indicates no limit.	-1
MaxDependentInPredicates	If IN criteria are supported, defines what the maximum number of predicates that can be used for a dependent join. Values less than 1 indicate to use only one IN predicate per dependent value pushed.	-1
NativeQueryProcedureName	If the native query is supported on the translator, this property indicates the name of the procedure.	native
SupportsNativeQueries	Set to true to indicate the translator supports the direct execution of commands using the **native** procedure	false

NOTE

Only a subset of the supports metadata can be set through execution properties. If more control is needed, see *Red Hat JBoss Data Virtualization Development Guide: Server Development*.

13.6. OVERRIDE EXECUTION PROPERTIES

You can override execution properties for any translator in the **`vdb.xml`** file:

```
<translator type="oracle-override" name="oracle">
     <property name="RequiresCriteria" value="true"/>
</translator>
```

The above XML fragment is overriding the oracle translator and altering the behavior of RequiresCriteria property setting it to true. Note that the modified translator is only available in the scope of this VDB.

13.7. PARAMETERIZABLE NATIVE QUERIES

In some situations the teiid_rel:native-query property and native procedures accept parameterizable strings that can positionally reference IN parameters. A parameter reference has the form $*integer*, for example, $1. Note that one-based indexing is used and that only IN parameters may be referenced. $*integer* is reserved, but may be escaped with another $, for example, $$1. The value will be bound as a prepared value or a literal in a source specific manner. The native query must return a result set that matches the expectation of the calling procedure.

For example, the native query "select c from g where c1 = $1 and c2 = '$$1'" results in a JDBC source query of "select c from g where c1 = ? and c2 = '$1'", where ? will be replaced with the actual value bound to parameter 1.

13.8. DELEGATING TRANSLATORS

You can create a delegating translator by extending the **`org.teiid.translator.BaseDelegatingExecutionFactory`** class.

Once your classes are packaged as a custom translator, you will be able to wire another translator instance into your delegating translator at runtime in order to intercept all of the calls to the delegate. This base class does not provide any functionality on its own, other than delegation.

Table 13.2. Execution Properties

Name	Description	Default
delegateName	Translator instance name to delegate to.	

As an example, consider if you are currently using "oracle" translator in your VDB and you need to intercept the calls going through this translator.

- You first write a custom delegating translator:

```
@Translator(name="interceptor", description="interceptor")
public class InterceptorExecutionFactory extends
org.teiid.translator.BaseDelegatingExecutionFactory{
    @Override
    public void getMetadata(MetadataFactory metadataFactory, C conn)
throws TranslatorException {
        // do intercepting code here..

        // If you need to call the original delegate, do not call if
do not need to.
        // but if you did not call the delegate fulfill the method
contract
        super.getMetadata(metadataFactory, conn);

        // do more intercepting code here..
    }
}
```

- Then you deploy this translator.
- Then modify your **`-vdb.xml`** or **`.vdb`** file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="myvdb" version="1">

    <model name="mymodel">
        <source name="source" translator-name="oracle-interceptor"
connection-jndi-name="java:oracle-ds"/>
    </model>

    <!-- the below it is called translator overriding, where you can
set different properties -->
    <translator name="oracle-interceptor" type="interceptor" />
        <property name="delegateName" value="oracle" />
   </translator>
</vdb>
```

We have defined a "translator" called "oracle-interceptor", which is based on the custom translator "interceptor" from above, and supplied the translator it required to delegate to "oracle" as its delegateName. Then, we used this override translator "oracle-interceptor" in the VDB. Now any calls going into this VDB model's translator will be intercepted by your code to do whatever you need to do.

13.9. AMAZON SIMPLEDB TRANSLATOR

Amazon SimpleDB is a web service for running queries on structured data in real time. This service works in close conjunction with Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2), collectively providing the ability to store, process and query data sets in the cloud. These services are designed to make web-scale computing easier and more cost-effective for developers.

This translator provides you with a way to connect to Amazon SimpleDB and it also provides relational functionality to add records directly from a user or from other sources that are integrated with Teiid. It does so via SQL. It also gives you the ability to read, update and delete existing records from the SimpleDB store.

Amazon SimpleDB is a hosted key/value store where a single key can contain multiple attribute name/value pairs where "value" can also be a multi-value.

When you import the metadata from SimpleDB into Teiid, the constructs are aligned like this:

Table 13.3. Registry Properties

Simple DB Name	SQL (Teiid)
Domain	Table
Item Name	Column (ItemName) Primary Key
attribute - single value	Column - String Datatype
attribute - multi value	Column - String Array Datatype

Since all attributes are considered, by default, to be string data types, columns are defined with string data type. However, during modeling of the schema in Designer, you can use various other data types supported through Teiid to define a data type of column, if you wish to expose one.

IMPORTANT

If you did modify the data type be something other than a string, do not use these changed columns in comparison queries, as SimpleDB does only lexicographical matching. To avoid using them, set the "SearchType" on the changed column to "UnSearchable"

This is an example Dynamic VDB that shows how you define the SimpleDB translator:

```
<vdb name="myvdb" version="1">
    <model name="simpledb">
        <source name="node" translator-name="simpledb" connection-jndi-
name="java:/simpledbDS"/>
    </model>
<vdb>
```

NOTE

The translator does not provide a connection to the SimpleDB. For that purpose, Teiid has a JCA adapter that provides a connection to SimpleDB using Amazon SDK Java libraries. To define such connector, see Amazon SimpleDB Data Sources or see an example in the **`jboss-as/docs/teiid/datasources/simpledb`** file.

If you are using Designer Tooling, to create a VDB, you can create or use a Teiid Designer Model project. Use the "Teiid Connection - Source Model" importer, create a SimpleDB Data Source using the Data Source Creation wizard and use simpledb as the translator in the importer. The table is created in a source model by this importer, providing that the data is already defined on Amazon SimpleDB. Create a VDB and deploy into Teiid Server and use either jdbc, odbc or odata to query it.

The Amazon SimpleDB Translator currently has no import or execution properties.

The Amazon SimpleDB Translator supports SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates, LIMIT and ORDER BY. Insert, update, delete are also supported.

Attributes with multiple values will defined as string array type. So this column is treated SQL Array type. This table shows the SimpleDB way of querying compared with the Teiid way of querying:

Table 13.4. Registry Properties

SimpleDB Query	Teiid Query
select * from mydomain where Rating = '4 stars' or Rating = '****'	select * from mydomain where Rating = ('4 stars','****')
select * from mydomain where Keyword = 'Book' and Keyword = 'Hardcover'	select * from mydomain where intersection(Keyword,'Book','Hardcover')
select * from mydomain where every(Rating) = '****'	select * from mydomain where every(Rating) = '****'

If you want to Insert/Update/Delete you can write prepare statements or you can compose SQL statements like this:

```
INSERT INTO mydomain (ItemName, title, author, year, pages, keyword,
rating) values ('0385333498', 'The Sirens of Titan', 'Kurt Vonnegut',
('1959'), ('Book', Paperback'), ('*****','5 stars','Excellent'))
```

WARNING

The Direct Query Support feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called SupportsDirectQueryProcedure to true.

NOTE

By default the name of the procedure that executes the queries directly is called native. Override the execution property DirectQueryProcedureName to change it to another name.

The SimpleDB translator provides a procedure to execute any ad-hoc simpledb query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications. Direct query supported for "select" based calls.

```
SELECT X.*
  FROM simpledb_source.native('SELECT firstname, lastname FROM users') n,
ARRAYTABLE(n.tuple COLUMNS firstname string, lastname string) AS X
```

The Teiid-specific Amazon SimpleDB Resource Adapter should be used with this translator

13.10. APACHE ACCUMULO TRANSLATOR

The Apache Accumulo Translator, known by the type name accumulo, exposes querying functionality to Accumulo Data Sources. Apache Accumulo is a sorted, distributed key value store with robust, scalable, high performance data storage and retrieval system. This translator provides an easy way connect to Accumulo system and provides relational way using SQL to add records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing records from Accumulo store. Teiid has capability to pass-in logged in user's roles as visibility properties to restrict the data access.

Here are some use cases for this translator:

- Accumulo source can be used in Teiid, to continually add/update the documents in the Accumulo system from other sources automatically.
- Access Accumulo through SQL interface.
- Make use of cell level security through enterprise roles.
- Accumulo translator can be used as an indexing system to gather data from other enterprise sources such as RDBMS, Web Service, SalesForce etc, all in single client call transparently with out any coding.

Apache Accumulo is distributed key value store with unique data model. It allows to group its key-value pairs in a collection called "table". You can define a schema representing Accumulo table structures in Teiid using DDL or using Teiid Designer with help of metadata extension properties defined below. Since no data type information is defined on the columns, by default all columns are considered as string data types. However, during modeling of the schema, one can use various other data types supported through Teiid to define a data type of column, that user wishes to expose.Once this schema is defined and exposed through VDB in a Teiid database, and Accumulo Data Sources is created, the user can issue "INSERT/UPDATE/DELETE" based SQL calls to insert/update/delete records into the Accumulo, and issue "SELECT" based calls to retrieve records from Accumulo. You can use full range of SQL with Teiid system integrating other sources along with Accumulo source.

IMPORTANT

By default, Accumulo table structure is flat and thus can not define relationships among tables. So, a SQL JOIN is performed in Teiid layer rather than pushed to source even if both tables on either side of the JOIN reside in the Accumulo. Currently any criteria based on EQUALITY and/or COMPARISON using complex AND/OR clauses are handled by Accumulo translator and will be properly executed at source.

Here is an example Dynamic VDB that shows the Accumulo translator:

```
<vdb name="myvdb" version="1">
    <model name="accumulo">
        <source name="node-one" translator-name="accumulo" connection-
```

```
jndi-name="java:/accumuloDS"/>
    </model>
<vdb>
```

The translator does NOT provide a connection to the Accumulo. For that purpose, Teiid has a JCA adapter that provides a connection to Accumulo using Accumulo Java libraries.

If you are using the Designer Tooling, to create a VDB create/use a Teiid Designer Model project, use the "Teiid Connection- Source Model" importer, create Accumulo Data Source using data source creation wizard and use accumulo as translator in the importer. The table is created in a source model by the time you finish with this importer. Create a VDB and deploy into Teiid Server and use either jdbc, odbc or odata to query.

The Accumulo translator is capable of traversing through Accumulo table structures and build a metadata structure for Teiid translator. The schema importer can understand simple tables by traversing a single ROWID of data, then looks for all the unique keys, based on it and comes up with a tabular structure for Accumulo based table. Using the following import properties, you can further refine the import behavior.

Table 13.5. Registry Properties

Property Name	Description	Required?	Default
ColumnNamePattern	How the column name is to be formed	false	{CF}_{CQ}
ValueIn	Where the value for column is defined CQ or VALUE	false	{VALUE}

NOTE

{CQ}, {CF}, {ROWID} are expressions that you can use to define above properties in any pattern, and respective values of Column Qualifer, Column Familiy or ROWID will be replaced at import time. ROW ID of the Accumulo table, is automatically created as ROWID column, and will be defined as Primary Key on the table.

You can also define the metadata for the Accumulo based model, using DDL or using the Teiid Designer. When doing such exercise, the Accumulo Translator currently defines following extended metadata properties to be defined on its Teiid schema model to guide the translator to make proper decisions. The following properties are described under NAMESPACE "http://www.teiid.org/translator/accumulo/2013", for user convenience this namespace has alias name teiid_accumulo defind in Teiid. To define a extension property use expression like "teiid_accumulo:{property-name} value". All the properties below are intended to be used as OPTION properties on COLUMNS. See DDL Metadata for more information on defining DDL based metadata.

Table 13.6. Registry Properties

Property Name	Description	Required?	Default
CF	Column Family	true	none
CQ	Column Qualifier	false	empty
VALUE-IN	Value of column defined in. Possible values (VALUE, CQ)	false	VALUE

Here is an example for a table called "User". A scan returns the following data:

```
root@teiid> table User
root@teiid User> scan
  1 name:age []    43
  1 name:firstname []   John
  1 name:lastname []    Does
  2 name:age []    10
  2 name:firstname []   Jane
  2 name:lastname []    Smith
  3 name:age []    13
  3 name:firstname []   Mike
  3 name:lastname []    Davis
```

If you used the default importer from the Accumulo translator(like Dynamic VDB defined above), the table generated will look like this:

```
CREATE FOREIGN TABLE "User" (
    rowid string OPTIONS (UPDATABLE FALSE, SEARCHABLE 'All_Except_Like'),
    name_age string OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'age',
"teiid_accumulo:VALUE-IN" '{VALUE}'),
    name_firstname string OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'firstname',
"teiid_accumulo:VALUE-IN" '{VALUE}'),
    name_lastname string OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'lastname',
"teiid_accumulo:VALUE-IN" '{VALUE}'),
    CONSTRAINT PK0 PRIMARY KEY(rowid)
) OPTIONS (UPDATABLE TRUE);
```

You can use "Import Property" as "ColumnNamePattern" as "{CQ}" will generate the following (note the names of the column):

```
CREATE FOREIGN TABLE "User" (
    rowid string OPTIONS (UPDATABLE FALSE, SEARCHABLE 'All_Except_Like'),
    age string OPTIONS (SEARCHABLE 'All_Except_Like', "teiid_accumulo:CF"
'name', "teiid_accumulo:CQ" 'age', "teiid_accumulo:VALUE-IN" '{VALUE}'),
    firstname string OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'firstname',
"teiid_accumulo:VALUE-IN" '{VALUE}'),
    lastname string OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_accumulo:CF" 'name', "teiid_accumulo:CQ" 'lastname',
"teiid_accumulo:VALUE-IN" '{VALUE}'),
    CONSTRAINT PK0 PRIMARY KEY(rowid)
) OPTIONS (UPDATABLE TRUE);
```

If the column name is defined by Column Family, you can use "ColumnNamePattern" as "{CF}", and if the value for that column exists in the Column Qualifier then you can use "ValueIn" as "{CQ}". Using import properties you can dictate how the table is to b= be modeled. If you did not use built in import (not using Teiid Designer's Teiid Connection >> Source Model or Dynamic VDB), and would like to manually design the table in Designer then you must make sure you supply the Extension Metadata Properties defined above on the User table's columns from Accumulo extended metadata(In Designer, right-click on Model, and select "Model Extension Definitions" and select Accumulo. For example on FirstName column, you would supply

The Teiid-specific Accumulo Resource Adapter must be used with this translator.

You cannot perform native queries or use direct query procedures with this translator.

13.11. APACHE SOLR TRANSLATOR

The Apache SOLR Translator, known by the type name solr, exposes querying functionality to Solr Data Sources. Apache Solr is a search engine built on top of Apache Lucene for indexing and searching. This translator provides an easy way connect to existing or a new Solr search system, and provides way to add documents/records from directly from user or from other sources that are integrated with Teiid. It also gives ability to read/update/delete existing documents from Solr Search system.

The Solr Translator currently has no import or execution properties. It does not define any extension metadata.

Here are some usecases for this translator:

- Solr source can be used in Teiid, to continually add/update the documents in the search system from other sources automatically.
- If the search fields are stored in Solr system, this can be used as very low latency data retrieval for serving high traffic applications.
- Solr translator can be used as a fast full text search. The Solr document can contain only the index information, then the results as an inverted index to gather target full documents from the other enterprise sources such as RDBMS, Web Service, SalesForce etc, all in single client call transparently with out any coding.

Solr search system provides searches based on indexed search fields. Each Solr instance is typically configured with a single core that defines multiple fields with different type information. Teiid metadata querying mechanism is equipped with "Luke" based queries, that at deploy time of the VDB use this mechanism to retrieve all the stored/indexed fields. Currently Teiid does NOT support dynamic fields and non-stored fields. Based on retrieved fields, Solr translator exposes a single table that contains all the fields. If a field is multi-value based, it's type is represented as Array type.

Once this table is exposed through VDB in a Teiid database, and Solr Data Sources is created, the user can issue "INSERT/UPDATE/DELETE" based SQL calls to insert/update/delete documents into the Solr, and issue "SELECT" based calls to retrieve documents from Solr. You can use full range of SQL with Teiid system integrating other sources along with Solr source.

The Solr Translator supports SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates, LIMIT and Order By.

Here is an example dynamic VDB that shows the Solr translator:

```
<vdb name="search" version="1">
    <model name="solr">
        <source name="node-one" translator-name="solr" connection-jndi-
name="java:/solrDS"/>
    </model>
<vdb>
```

The translator does NOT provide a connection to the Solr. For that purpose, Teiid has a JCA adapter that provides a connection to Solr using the SolrJ Java library. See an example in see an example in **`jboss-as/docs/teiid/datasources/solr`**

If you are using Designer Tooling, to create VDB then create/use a Teiid Designer Model project, u Use "Teiid Connection - Source Model" importer, create Solr Data Source using data source creation wizard and use solr as translator in the importer. The search table is created in a source model by the time you finish with this importer. Create a VDB and deploy into Teiid Server and use either jdbc, odbc or odata to query.

The Teiid specific Solr Resource Adapter should be used with this translator.

13.12. CASSANDRA TRANSLATOR

WARNING

The Cassandra Translator is a technology preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), may not be functionally complete, and are not recommended to be used for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

The Cassandra Translator, known by the type name cassandra, exposes querying functionality to Cassandra Data Sources. The translator translates Teiid push down commands into Cassandra CQL.

The Cassandra Translator currently has no import or execution properties.

The Cassandra Translator supports only SELECT statements with a restrictive set of capabilities including: count(*), comparison predicates, IN predicates, and LIMIT. Consider a custom extension or create an enhancement request should your usage require additional capabilities.

The Teiid-specific Cassandra Resource Adapter should be used with this translator.

Cassandra source procedures may be created using the teiid_rel:native-query extension. The procedure will invoke the native-query similar to a direct procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

WARNING

The direct query procedure feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called SupportsDirectQueryProcedure to true.

By default the name of the procedure that executes the queries directly is called native. Override the execution property DirectQueryProcedureName to change it to another name.

The Cassandra translator provides a procedure to execute any ad-hoc CQL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array. ARRAYTABLE can be used construct tabular output for consumption by client applications.

```
SELECT X.*
  FROM cassandra_source.native('SELECT firstname, lastname FROM users
WHERE birth_year = $1 AND country = $2 ALLOW FILTERING', 1981, 'US') n,
       ARRAYTABLE(n.tuple COLUMNS firstname string, lastname string) AS X
```

13.13. FILE TRANSLATOR

13.13.1. File Translator

The file translator exposes stored procedures to leverage file system resources exposed by the file resource adapter. It will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formatted data. See Section 3.6.7, “Nested Tables: TEXTTABLE” and Section 3.6.8, “Nested Tables: XMLTABLE”.

The file translator is implemented by the `org.teiid.translator.file.FileExecutionFactory` class and known by the translator type name `file`.

NOTE

The resource adapter for this translator is provided by configuring the `file` data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.13.2. File Translator: Execution Properties

Table 13.7. Execution Properties

Name	Description	Default
Encoding	The encoding that must be used for CLOBs returned by the getTextFiles procedure.	The system default encoding
ExceptionIfFileNotFound	Throw an exception in getFiles or getTextFiles if the specified file/directory does not exist.	true (false in previous releases)

13.13.3. File Translator: Usage

Retrieve all files as BLOBs with an optional extension at the given path.

```
call getFiles('path/*.ext')
```

If the extension path is specified, then it will filter all of the files in the directory referenced by the base path. If the extension pattern is not specified and the path is a directory, then all files in the directory will be returned. Otherwise the single file referenced will be returned. If the path does not exist, then no results will be returned if **`ExceptionIfFileNotFound`** is false, otherwise an exception will be raised.

Retrieve all files as CLOB(s) with the an optional extension at the given path.

```
call getTextFiles('path/*.ext')
```

`getTextFiles` will retrieve the same files as **`getFiles`**, only the results will be CLOB values using the encoding execution property as the character set.

Save the CLOB, BLOB, or XML value to the given path.

```
call saveFile('path', value)
```

The path is a reference to either a new file location or an existing file to overwrite completely.

NOTE

Native or direct query execution is not supported on the File Translator.

13.14. GOOGLE SPREADSHEET TRANSLATOR

13.14.1. Google Spreadsheet Translator

The Google spreadsheet translator is used to connect to a Google spreadsheet.

The Google spreadsheet translator is implemented by the **`org.teiid.translator.google.SpreadsheetExecutionFactory`** class and known by the translator type name **`google-spreadsheet`**.

The query approach expects the data in the worksheet to be in a specific format. Namely:

- Any column that has data can be queried.
- Any non-string column with an empty cell has the value retrieved as null, otherwise it is the empty string.
- If the first row is present and contains string values, then it will be assumed to represent the column labels.

If you are using a dynamic VDB, the metadata for your Google account (worksheets and information about columns in worksheets) are loaded upon translator start up. If you make any changes in data types, it is advisable to restart your VDB.

The translator supports queries against a single sheet. It supports ordering, aggregation, basic predicates, and most of the functions supported by the spreadsheet query language.

There are no Google spreadsheet importer settings, but it does provide metadata for dynamic VDBs.

NOTE

The resource adapter for this translator is provided by configuring the `google` data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.14.2. Google Spreadsheet Translator: Native Queries

Google spreadsheet source procedures may be created using the teiid_rel:native-query extension (see Section 13.7, “Parameterizable Native Queries”) The procedure will invoke the native query similar to an native procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE (Section 3.6.9, “Nested Tables: ARRAYTABLE”) or similar functionality.

13.14.3. Google Spreadsheet Translator: Native Procedure

WARNING

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the translator property called "SupportsNativeQueries" to true. See Section 13.6, “Override Execution Properties”.

Google spreadsheet translator provides a procedure with name **native** that gives ability to execute any ad hoc native Google spreadsheet queries directly against the source without any JBoss Data Virtualization parsing or resolving. Since the metadata of this procedure's execution results are not known to the JBoss Data Virtualization and they are returned as object array. Users can use ARRAYTABLE (Section 3.6.9, “Nested Tables: ARRAYTABLE”) to construct a build a tabular output for consumption by client applications.

JBoss Data Virtualization exposes this procedure with a simple query structure:

Example 13.1. Select Example

```
SELECT x.* FROM (call pm1.native('worksheet=People;query=SELECT A, B,
C')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String)
AS x
```

The first argument takes semi-colon(;) separated name value pairs of following properties to execute the procedure:

Property	Description	Required
worksheet	Google spreadsheet name	yes
query	spreadsheet query	yes
limit	number rows to fetch	no
offset	offset of rows to fetch from limit or beginning	no

NOTE

By default the name of the procedure that executes the queries directly is called **native** , however the user can set the Override Execution Properties property (see Section 13.6, "Override Execution Properties") on NativeQueryProcedureName in the **vdb.xml** file to change it to any other procedure name.

13.15. INFINISPAN DSL TRANSLATOR

The Infinispan DSL Translator, known by the type infinispan-cache-dsl, can read the Java objects from a remote Infinispan Cache via the Hot Rod client using the Google Protobuf for serialization. The benefit of this JBoss Data Grid design is that it will enable Teiid to query the cache using DSL, which is similar to doing Lucene searching on a local cache. If you are using Infinispan in Library mode, see the Object Translator for this type of configuration.

The Infinispan DSL Translator currently has no import or execution properties. See the JCA resource adapter section below for information on how to configure the cache to be queried.

Here are the connector's capabilities:

- Compare Criteria - EQ, NE, LT, GT, LE, GE.
- And/Or Criteria
- (Not) In Criteria
- (Not) Like Criteria
- (Not) IsNull

- INSERT, UPDATE, DELETE (non-transactional)

Here are its limitations:

- One-to-Many, currently only supports Collection or Array, not Maps
- Write transactions not supported by JDG when using Hot Rod client

Use it to retrieve objects from a cache and transform into rows and columns and to perform writes to the cache.

Here are the definition requirements:

- Each Google registered class in the cache will have a corresponding table created.
- The table for the root class, must have a primary key defined, which must map to an attribute in the class.
- The table "name in source" (NIS) will be the name of the JDG cache this table/class is stored
- The table columns will be created from the Google protobuf definition, that corresponds to a registered class.
- Attributes defined as repeatable (i.e., collections, arrays, etc.) or a container class, will be supported as 1-to-* relationships, and will have corresponding registered class (if they are to be searched).
- A one-to-many relationship class must have a foreign key to map to the root class/table, where the name in source for the foreign key is the name of the root class method to access those child objects. Note, this is the class method, not a reference in the Google protobuf definition.
- A container/child class will have attributes where the NIS contain a period. Example: phone.number. This is because this maps to the Google protobuf definition and what is expected to be used in the DSL query.

There are several options to defining the metadata representing your object in the cache.

- Use dynamic VDB that only defines the data source to use. The metadata will be resolved by reverse engineering the defined object in the cache. This can be useful when using the Teiid Designer Teiid Connection Importer for building the physical source model(s).

```
<model name="People" type="Physical">
    <property name="importer.useFullSchemaName" value="false"/>

    <source name="infinispan-cache-dsl-connector" translator-
name="infinispan-cache-dsl" connection-jndi-
name="java:/infinispanRemoteDSL" />
</model>
```

- Use Teiid Designer to manually create the physical source model based on your object cache using the above Usage patterns.

See the Infinispan-DSL resource adapter for this translator. It can be configured to lookup the cache container via JNDI, server list, or hot rod properties.

IMPORTANT

CompareCriteriaOrdered has been disabled due to a Red Hat JBoss Data Grid issue with filtering when GTE/LTE criteria is specified on String type attributes. This criteria is not sent to JBoss Data Grid for processing, but is processed in Teiid. As a result, a performance issue can arise with these types of queries, depending on what other criteria has also been specified.

If you wish to enable this capability so that GTE/LTE can be used on other data types, then specify a translator override for CompareCriteriaOrdered. Here is how you define an override in a dynamic VDB:

```
<translator name="infinispan-cache-dsl1" type="infinispan-
cache-dsl">
  <property name="supportsCompareCriteriaOrdered"
value="true"/>
</translator>
```

IMPORTANT

Note that char types are not supported when accessing a JBoss Data Grid Remote Cache and using Google Protobufs for serialization. To see which data types are supported, see this table: https://developers.google.com/protocol-buffers/docs/proto#scalar

To work around the current limitation on the Char type, use the String data type or use the marshaller to handle the conversion.

13.16. JDBC TRANSLATOR

13.16.1. JDBC Translator

The JDBC translator bridges between SQL semantic and data type difference between JBoss Data Virtualization and a target RDBMS.

The base JDBC translator is implemented by the `org.teiid.translator.jdbc.JDBCExecutionFactory` class.

NOTE

The resource adapter for a particular JDBC translator is provided by configuring the corresponding data source in teh JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.16.2. JDBC Translator: Execution Properties

The following execution properties are shared by all JDBC translators.

Table 13.8. Execution Properties

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetching date, time, or timestamp values.	The system default time zone
DatabaseVersion	The specific database version. Used to further tune pushdown support.	The base supported version or derived from the `DatabaseMetadata.getProduceVersion` string. Automatic detection requires a Connection. If there are circumstances where you are getting an exception from capabilities being unavailable (most likely due to an issue obtaining a Connection), then set the `DatabaseVersion` property. Use the `JDBCExecutionFactory.usesDatabaseVersion()` method to control whether your translator requires a connection to determine capabilities.
TrimStrings	Set to true to trim trailing whitespace from fixed length character strings. Note that JBoss Data Virtualization only has a string, or varchar, type that treats trailing whitespace as meaningful.	false
UseBindVariables	Set to true to indicate that PreparedStatements will be used and that literal values in the source query will be replaced with bind variables. If false, only LOB values will trigger the use of PreparedStatements.	true
UseCommentsInSourceQuery	This will embed a leading comment with session/request id in source SQL query for informational purposes	false

Name	Description	Default
CommentFormat	MessageFormat string to be used if UseCommentsInSourceQuery is enabled. Available properties: <ul><li>0 - session id string</li><li>1 - parent request id string</li><li>2 - request part id string</li><li>3 - execution count id string</li><li>4 - user name string</li><li>5 - vdb name string</li><li>6 - vdb version integer</li><li>7 - is transactional boolean</li></ul>	`/*teiid sessionid:{0}, requestid:{1}.{2}*/`
MaxPreparedInsertBatchSize	The max size of a prepared insert batch.	2048
StructRetrieval	Struct retrieval mode can be one of OBJECT - getObject value returned, COPY - returned as a SerialStruct, ARRAY - returned as an Array)	OBJECT
EnableDependentJoins	For sources that support temporary tables (DB2, Derby, H2, HSQL 2.0+, MySQL 5.0+, Oracle, PostgreSQL, SQLServer, Sybase) allow dependent join pushdown	false

13.16.3. JDBC Translator: Importer Properties

The following properties are shared by all JDBC translators.

Table 13.9. Importer Properties

Name	Description	Default
catalog	See DatabaseMetaData.getTables at http://download.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html for more information.	null

Name	Description	Default
schemaPattern	See DatabaseMetaData.getTables at http://download.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html for more information.	null
tableNamePattern	See DatabaseMetaData.getTables at http://download.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html for more information.	null
procedureNamePattern	See DatabaseMetaData.getProcedures at http://download.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html for more information.	null
tableTypes	Comma separated list - without spaces - of imported table types. See DatabaseMetaData.getTables at http://download.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html for more information.	null
excludeTables	A case-insensitive regular expression that when matched against a fully qualified JBoss Data Virtualization table name will exclude it from import. Applied after table names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	null
excludeProcedures	A case-insensitive regular expression that when matched against a fully qualified JBoss Data Virtualization procedure name will exclude it from import. Applied after procedure names are retrieved. Use a negative look-ahead (?!<inclusion pattern>).* to act as an inclusion filter.	null
autoCreateUniqueConstraints	True to create a unique constraint if one is not found for a foreign keys	true
useFullSchemaName	When false, directs the importer to drop the source catalog/schema from the JBoss Data Virtualization object name, so that the JBoss Data Virtualization fully qualified name will be in the form of <model name>.<table name>. Note that when this is false, it may lead to objects with duplicate names when importing from multiple schemas, which results in an exception. This option does not affect the name in source property.	true
importKeys	Set to true to import primary and foreign keys.	true
importIndexes	Set to true to import index/unique key/cardinality information.	false

Name	Description	Default
importApproximateIndexes	Set to true to import approximate index information. See DatabaseMetaData.getIndexInfo at http://download.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html for more information.	true
importProcedures	Set to true to import procedures and procedure columns. Note that it is not always possible to import procedure result set columns due to database limitations. It is also not currently possible to import overloaded procedures.	true
widenUnsignedTypes	Set to true to convert unsigned types to the next widest type. For example SQL Server reports tinyint as an unsigned type. With this option enabled, tinyint would be imported as a short instead of a byte.	true
quoteNameInSource	Set to false to override the default and direct JBoss Data Virtualization to create source queries using unquoted identifiers.	true
useProcedureSpecificName	Set to true to allow the import of overloaded procedures (which will normally result in a duplicate procedure error) by using the unique procedure-specific name as the JBoss Data Virtualization name. This option will only work with JDBC 4.0 compatible drivers that report specific names.	false
useCatalogName	Set to true to use any non-null/non-empty catalog name as part of the name in source, e.g. "catalog"."table"."column", and in the JBoss Data Virtualization runtime name if useFullSchemaName is true. Set to false to not use the catalog name in either the name in source or the JBoss Data Virtualization runtime name. Must be set to false for sources that do not fully support a catalog concept, but return a non-null catalog name in their metadata, such as HSQL.	true
useQualifiedName	True will use name qualification for both the Teiid name and name in source as dictated by the useCatalogName and useFullSchemaName properties. Set to false to disable all qualification for both the Teiid name and the name in source, which effectively ignores the useCatalogName and useFullSchemaName properties. Note: when false this may lead to objects with duplicate names when importing from multiple schemas, which results in an exception.	true

Name	Description	Default
useAnyIndexCardinality	True will use the maximum cardinality returned from DatabaseMetaData.getIndexInfo. importKeys or importIndexes needs to be enabled for this setting to have an effect. This allows for better stats gathering from sources that do not support returning a statistical index.	false
importStatistics	This uses database-dependent logic to determine the cardinality if none is determined. (This is currently only supported on Oracle and MySQL.)	false

WARNING

The default import settings will traverse all available metadata. This import process is time consuming and full metadata import is not needed in most situations. In most situations you will limit import by at least schemaPattern and tableTypes.

Example importer settings to only import tables and views from my-schema.

```
 <model ...

  <property name="importer.tableTypes" value="TABLE,VIEW"/>
  <property name="importer.schemaPattern" value="my-schema"/>
  ...
</model>
```

13.16.4. JDBC Translator: Translator Types

JBoss Data Virtualization has a range of specific translators that target the most popular open source and proprietary databases.

jdbc-ansi

This translator provides support for most SQL constructs supported by JBoss Data Virtualization, except for row limit/offset and EXCEPT/INTERSECT. It translates source SQL into ANSI compliant syntax.

This translator can be used when another more specific type is not available.

jdbc-simple

This translator is the same as jdbc-ansi, except that it disables support for function, UNION and aggregate pushdown.

access

This translator is for use with Microsoft Access 2003 or later.

db2

This translator is for use with DB2 8 or later (and DB2 for i 5.4 or later).

Execution properties specific to DB2:

- *DB2ForI* indicates that the DB2 instance is DB2 for i. Defaults to false.

Apache HBase Translator

The Apache HBase Translator exposes querying functionality to HBase Tables. Apache Phoenix is an SQL interface for HBase. With the Phoenix Data Sources, the translator translates Teiid push-down commands into Phoenix SQL.

The HBase Translator does not support Join commands, because Phoenix has more simple constraints. The only supported is that for the Primary Key, which maps to the HBase Table Row ID. This translator is developed with Phoenix 4.x for HBase 0.98.1+.

If you use the translator for Apache HBase, be aware that insert statements can rewrite data. To illustrate, here is a standard set of SQL queries:

```
CREATE TABLE TableA (id integer PRIMARY KEY, name varchar(10));
INSERT INTO TableA (id, name) VALUES (1, 'name1');
INSERT INTO TableA (id, name) VALUES (1, 'name2');
```

Normally, the second INSERT command would fail as the uniqueness of the primary key would be corrupted. However, with the HBase translator, the command will not fail. Rather, it will rewrite the data in the table, (so "name1" would become "name2"). This is because the translator converts the INSERT command into an UPSERT command.

Derby

derby - for use with Derby 10.1 or later.

excel-odbc

This translator is for use with Excel 2003 or later via a JDBC-ODBC bridge.

greenplum

This translator is for use with the Greenplum database.

h2

For use with h2 version 1.1 or later.

hive

This translator is for use with the Hive database. Hive is a data warehousing infrastructure based on Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing (using the map-reduce programming paradigm) on commodity hardware.

Hive has limited support for data types as it supports integer variants, boolean, float, double and string. It does not have native support for time based types, xml or LOBs. These limitations are reflected in the translator capabilities. The view table can use these types, however the translator would need to specify the necessary transformations. Note that in those situations, the evaluations will be done in the JBoss Data Virtualization engine. Another limitation Hive has is, it only supports

EQUI join, so using any other join types on its source tables will result in inefficient queries. Currently there is no tooling support for metadata import from Hive in Teiid Designer. Partition columns may be used as criteria within a WHERE clause, however, they cannot be returned as part of the result set. The Hive importer does not have concept of catalog or source schema, nor does it import keys, procedures, indexes and so forth.

Importer properties specific to Hive:

- *trimColumnNames*: For Hive 0.11.0 and later the DESCRIBE command metadata is returned with padding. Set to true to strip white space from column names. Defaults to false.

IMPORTANT

The Hive importer does not currently use typical JDBC DatabaseMetaData calls, nor does it have the concept of catalog or source schema, nor does it import keys, procedures, indexes, etc. Thus not all of the common JDBC importer properties are applicable to Hive. You can still use excludeTables.

useDatabaseMetaData: For Hive 0.13.0 and later the normal JDBC DatabaseMetaData facilities are sufficient to perform an import. Set to true to use the normal import logic with the option to import index information disabled. Defaults to false. When true, trimColumnNames has no effect. If false the typical JDBC DatabaseMetaData calls are not used so not all of the common JDBC importer properties are applicable to Hive. You canstill use excludeTables regardless.

IMPORTANT

When the database name used in the Hive is differs from "default", the metadata retrieval and execution of queries does not work as expected in Teiid, as Hive JDBC driver seems to be implicitly connecting (tested with versions lower than 0.12) to "default" database, thus ignoring the database name mentioned on connection URL. You can work around this in Red Hat JBoss Data Virtualization in the JBoss EAP environment by setting the following in data source configuration:

```
<new-connection-sql>use {database-name}</new-
connection-sql>
```

This is fixed in version 0.13 and later of the Hive Driver.

hsql

This translator is for use with HSQLDB 1.7 or later.

impala

This translator is for use with Cloudera Impala 1.2.1 or later.

Impala has limited support for data types. It is does not have native support for time/date/xml or LOBs. These limitations are reflected in the translator capabilities. A Teiid view can use these types, however the transformation would need to specify the necessary conversions. Note that in those situations, the evaluations will be done in Teiid engine.

Impala only supports EQUI join, so using any other joins types on its source tables will result in inefficient queries.

To write criteria based on partitioned columns, model them on the source table, but do not include them in selection columns.

IMPORTANT

The Impala importer does not currently use typical JDBC DatabaseMetaData calls, nor does it have the concept of catalog or source schema, nor does it import keys, procedures, indexes, etc. Thus not all of the common JDBC importer properties are applicable to Impala. You may still use excludeTables.

Impala specific importer properties:

useDatabaseMetaData - Set to true to use the normal import logic with the option to import index information disabled. Defaults to false.

If false the typical JDBC DatabaseMetaData calls are not used so not all of the common JDBC importer properties are applicable to Impala. (You can still use excludeTables regardless.)

IMPORTANT

Some versions of Impala require the use of a LIMIT when performing an ORDER BY. If no default is configured in Impala, an exception can occur when a Teiid query with an ORDER BY but no LIMIT is issued. You must set an Impala wide default, or configure the connection pool to use a new connection SQL string to issue a SET DEFAULT_ORDER_BY_LIMIT statement. See the Cloudera documentationfor more on limit options, such as controlling what happens when the limit is exceeded.

ingres

This translator is for use with Ingres 2006 or later.

ingres93

This translator is for use with Ingres 9.3 or later.

intersystems-cache

For use with Intersystems Cache Object database (only relational aspect of it)

informix

For use with any Informix version.

metamatrix

This translator is for use with MetaMatrix 5.5.0 or later.

modeshape

This translator is for use with Modeshape 2.2.1 or later.

The PATH, NAME, LOCALNODENAME, DEPTH, and SCORE functions are accessed as pseudo-columns, e.g. "nt:base"."jcr:path".

JBoss Data Virtualization user defined functions (prefixed by JCR_) are available for CONTAINS, ISCHILDNODE, ISDESCENDENT, ISSAMENODE, REFERENCE. See the `JCRFunctions.xmi` file.

If a selector name is needed in a JCR function, you can use the pseudo-column "jcr:path". For example, JCR_ISCHILDNODE(foo.jcr_path, 'x/y') would become ISCHILDNODE(foo, 'x/y') in the ModeShape query.

An additional pseudo-column "mode:properties" can be imported by setting the ModeShape JDBC connection property teiidsupport=true. The "mode:properties" column should be used by the JCR_REFERENCE and other functions that expect a .* selector name. For example, JCR_REFERENCE(nt_base.jcr_properties) would become REFERENCE("nt:base".*) in the ModeShape query.

mysql5

This translator is for use with MySQL version 5 or later. It also works with backwards-compatible MySQL derivatives, including MariaDB.

The MySQL Translator expects the database or session to be using ANSI mode. If the database is not using ANSI mode, an initialization query must be used on the pool to set ANSI mode:

```
set SESSION sql_mode = 'ANSI'
```

If you deal with null timestamp values, then set the connection property zeroDateTimeBehavior=convertToNull. Otherwise you'll get conversion errors in Teiid that '0000-00-00 00:00:00' cannot be converted to a timestamp.

netezza

This translator is for use with any Netezza version.

IMPORTANT

The current Netezza vendor supplied JDBC driver performs poorly with single transactional updates. As is generally the case, use batched updates when possible.

Netezza-specific execution properties:

SqlExtensionsInstalled- indicates that SQL Extensions including support fo REGEXP_LIKE are installed. Defaults to false.

oracle

This translator is for use with Oracle 9i or later.

Sequences may be used with the Oracle translator. A sequence may be modeled as a table with a name in source of DUAL and columns with the name in source set to this:

```
      {{<sequence name>.[nextval|currval].}}</code>
```

Teiid 8.4 and Prior Oracle Sequence DDL

```
CREATE FOREIGN TABLE seq (nextval integer OPTIONS (NAMEINSOURCE
'seq.nextval'), currval integer options (NAMEINSOURCE 'seq.currval') )
OPTIONS (NAMEINSOURCE 'DUAL')
```

With Teiid 8.5 it is no longer necessary to rely on a table representation and Oracle specific handling for sequences. See DDL Metadata for representing currval and nextval as source functions.

You can also use a sequence as the default value for insert columns by setting the column to autoincrement and the name in source to this:

```
<element name>:SEQUENCE=<sequence name>.<sequence value>.
```

A rownum column can be added to any Oracle physical table to support the rownum pseudo-column. A rownum column must have a name in source of this:

```

     <code>rownum</code>.

```

These rownum columns do not have the same semantics as the Oracle rownum construct so care must be taken in their usage.

Oracle specific importer properties:

useGeometryType- Use the Teiid Geomety type when importing columns with a source type of SDO_GEOMETRY. Defaults to false.

useIntegralTypes- Use integral types rather than decimal when the scale is 0. Defaults to false.

Execution properties specific to Oracle:

- *OracleSuppliedDriver* - indicates that the Oracle supplied driver (typically prefixed by ojdbc) is being used. Defaults to true. Set to false when using DataDirect or other Oracle JDBC drivers.

postgresql

This translator is for use with 8.0 or later clients and 7.1 or later server.

PostgreSQL specific execution properties:

PostGisVersion- indicate the PostGIS version in use. Defaults to 0 meaning PostGIS is not installed. Will be set automatically if the database version is not set. _ProjSupported- boolean indicating if Proj is support for PostGis. Will be set automatically if the database version is not set.

prestodb

The PrestoDB translator, known by the type name prestodb, exposes querying functionality to PrestoDB Data Sources. In data integration respect, PrestoDB has very similar capabilities of Teiid, however it goes beyond in terms of distributed query execution with multiple worker nodes. Teiid's execution model is limited to single execution node and focuses more on pushing the query down to sources. Currently Teiid has much more complete query support and many enterprise features.

The PrestoDB translator supports only SELECT statements with a restrictive set of capabilities. This translator is developed with 0.85 version of PrestoDB and capabilities are designed for this version. With new versions of PrestoDB Teiid will adjust the capabilities of this translator. Since PrestoDB exposes a relational model, the usage of this is no different than any RDBMS source like Oracle, DB2 etc. For configuring the PrestoDB consult the PrestoDB documentation.

sqlserver

This translator is for use with SQL Server 2000 or later. A SQL Server JDBC driver version 2.0 or later (or compatible e.g. JTDS 1.2 or later) must be used. The SQL Server `DatabaseVersion` property may be set to 2000, 2005, 2008, or 2012, but otherwise expects a standard version number, for example, 10.0.

Execution properties specific to SQL Server:

- *JtdsDriver* - indicates that the open source JTDS driver is being used. Defaults to false.

sybase

This translator is for use with Sybase version 12.5 or later. If used in a dynamic vdb and no import properties are specified (not recommended, see import properties below), then exceptions can be thrown retrieving system table information. Specify a schemaPattern or use excludeTables to exclude system tables if this occurs.

If the name in source metadata contains quoted identifiers (such as reserved words or words containing characters that would not otherwise be allowed) and you are using a jconnect Sybase driver, you must first configure the connection pool to enable **`quoted_identifier`**.

Example 13.2. Driver URL with SQLINITSTRING

```
jdbc:sybase:Tds:host.at.some.domain:5000/db_name?SQLINITSTRING=set
quoted_identifier on
```

Execution properties specific to Sybase:

- *JtdsDriver* - indicates that the open source JTDS driver is being used. Defaults to false.

IMPORTANT

You must set the connection parameter JCONNECT_VERSION to 6 or later when using the Sybase data source. If you do not do so, you will encounter an exception.

sybaseiq

This translator is for use with Sybase IQ version 15.1 or later.

teiid

This translator is for use with Teiid 6.0 or later.

teradata

This translator is for use with Teradata V2R5.1 or later.

13.16.5. JDBC Translator: Usage

Using JBoss Data Virtualization SQL, the source may be queried as if the tables and procedures were local to the JBoss Data Virtualization system.

13.16.6. JDBC Translator: Native Queries

Both physical tables and procedures may optionally have native queries associated with them. No validation of the native query is performed; it is used to generate the source SQL.

For a physical table, setting the teiid_rel:native-query extension metadata to the desired query string will execute the native query as an inline view in the source query. This feature can only be used against sources that support inline views. The native query is used as is and is not treated as a parameterized string. For example, on a physical table y with nameInSource="x" and teiid_rel:native-query="select c from g", the JBoss Data Virtualization source query "SELECT c FROM y" would generate the SQL query "SELECT c FROM (select c from g) as x". Note that the column names in the native query must match the nameInSource of the physical table columns for the resulting SQL to be valid.

For physical procedures, you may also set the teiid_rel:native-query extension metadata to a desired query string with the added ability to positionally reference IN parameters (see Section 13.7, “Parameterizable Native Queries”).

A parameter reference has the form $*integer*, for example, $1. Note that one-based indexing is used and that only IN parameters may be referenced. $*integer* is reserved, but may be escaped with another $, for example, $$1.

By default, bind values will be used for parameter values. In some situations you might need to bind values directly into the resulting SQL.

The teiid_rel:non-prepared extension metadata property may be set to false to turn off parameter binding. Note that this option must be used with caution as inbound may allow for SQL injection attacks if not properly validated. The native query does not need to call a stored procedure. Any SQL that returns a result set positionally matching the result set expected by the physical stored procedure metadata will work. For example, on a stored procedure x with teiid_rel:native-query="select c from g where c1 = $1 and c2 = '$$1'", the JBoss Data Virtualization source query "CALL x(?)" would generate the SQL query "select c from g where c1 = ? and c2 = '$1'". Note that ? in this example will be replaced with the actual value bound to parameter 1.

13.16.7. JDBC Translator: Native Procedure

WARNING

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the translator property called "SupportsNativeQueries" to true. See Section 13.6, “Override Execution Properties”.

JDBC translator also provides a procedure with name **native** that gives ability to execute any ad hoc native SQL command that is specific to an underlying source directly against the source without any JBoss Data Virtualization parsing or resolving. The metadata of this procedure's execution results are not known to JBoss Data Virtualization, and they are returned as object array. Users can use the ARRAYTABLE construct (Section 3.6.9, “Nested Tables: ARRAYTABLE”) to produce tabular output for client applications.

Example 13.3. Select Example

```
SELECT x.* FROM (call pm1.native('select * from g1')) w,
 ARRAYTABLE(w.tuple COLUMNS "e1" integer , "e2" string) AS x
```

Example 13.4. Insert Example

```
SELECT x.* FROM (call pm1.native('insert into g1 (e1,e2) values (?, ?)',
112, 'foo')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Example 13.5. Update Example

```
SELECT x.* FROM (call pm1.native('update g1 set e2=? where e1 =
?','blah', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

Example 13.6. Delete Example

```
SELECT x.* FROM (call pm1.native('delete from g1 where e1 = ?', 112)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

IMPORTANT

By default, the name of the procedure that executes the queries directly is called **native** , however users can override the NativeQueryProcedureName execution property in the **`vdb.xml`** file to change it to any other procedure name. See Section 13.6, “Override Execution Properties”.

13.17. JPA TRANSLATOR

The JPA translator, known by the type name jpa2, can reverse a JPA object model into a relational model, which can then be integrated with other relational or non-relational sources.

The JPA Translator currently has no import or execution properties.

JPA source procedures may be created using the teiid_rel:native-query extension. The procedure invokes the native-query similar to an native procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

WARNING

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, set the execution property called SupportsDirectQueryProcedure to true.

NOTE

By default the name of the procedure that executes the queries directly is native. Override the execution property DirectQueryProcedureName to change it to another name.

The JPA translator provides a procedure to execute any ad-hoc JPA-QL query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as object array. User can use ARRAYTABLE can be used construct tabular output for consumption by client applications. Teiid exposes this procedure with a query structure.

In this select query, the "search" keyword is followed by a query statement:

```
SELECT x.* FROM (call jpa_source.native('search;FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String) AS
x
```

In this delete query, the the "delete" keyword is followed by JPA-QL for a delete operation.

```
SELECT x.* FROM (call jpa_source.native('delete;<jpa-ql>')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

In this sample, the "update" keyword must be followed by JPA-QL for the update statement.

```
SELECT x.* FROM
 (call jpa_source.native('update;<jpa-ql>')) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

In this create query, the create operation sends "create" word as a marker and send the entity as the first parameter:

```
SELECT x.* FROM
 (call jpa_source.native('create;', <entity>)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

13.18. LDAP TRANSLATOR

13.18.1. LDAP Translator

The LDAP translator exposes an LDAP directory tree relationally with pushdown support for filtering via criteria. This is typically coupled with the LDAP resource adapter.

The LDAP translator is implemented by the `org.teiid.translator.ldap.LDAPExecutionFactory` class and known by the translator type name `ldap`.

NOTE

The resource adapter for this translator is provided by configuring the `ldap` data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.18.2. LDAP Translator: Execution Properties

Table 13.10. Execution Properties

Name	Description	Default
SearchDefaultBaseDN	Default Base DN for LDAP Searches	null
SearchDefaultScope	Default Scope for LDAP Searches. Can be one of SUBTREE_SCOPE, OBJECT_SCOPE, ONELEVEL_SCOPE.	ONELEVEL_SCOPE
RestrictToObjectClass	Restrict Searches to objectClass named in the Name field for a table	false
UsePagination	Use a PagedResultsControl to page through large results. This is not supported by all directory servers.	false
ExceptionOnSizeLimitExceeded	Set to true to throw an exception when a SizeLimitExceededException is received and a LIMIT is not properly enforced.	false

NOTE

There are no import settings for the LDAP translator; it also does not provide metadata.

If one of the methods below is not used and the attribute is mapped to a non-array type, then any value may be returned on a read operation. Also insert/update/delete support will not be multi-value aware.

String columns with a default value of "multivalued-concat" will concatenate all attribute values together in alphabetical order using a ? delimiter. If a multivalued attribute does not have a default value of "multivalued-concat", then any value may be returned.

Multiple attribute values may also be supported as an array type. The array type mapping also allows for insert/update operations.

This example shows a DDL with objectClass and uniqueMember as arrays:

```
create foreign table ldap_groups (objectClass string[], DN string, name
string options (nameinsource 'cn'), uniqueMember string[]) options
(nameinsource 'ou=groups,dc=teiid,dc=org', updatable true)
```

The array values can be retrieved with a SELECT. Here is an example insert with array values:

```
insert into ldap_groups (objectClass, DN, name, uniqueMember) values
(('top', 'groupOfUniqueNames'), 'cn=a,ou=groups,dc=teiid,dc=org', 'a',
('cn=Sam Smith,ou=people,dc=teiid,dc=org',))
```

13.18.3. LDAP Translator: Metadata Directives

String columns with a default value of "multivalued-concat" will concatenate all attribute values together in alphabetical order using a ? delimiter. If a multivalued attribute does not have a default value of "multivalued-concat", then any value may be returned.

13.18.4. LDAP Translator: Native Queries

LDAP procedures may optionally have native queries associated with them (see Section 13.7, “Parameterizable Native Queries”). The operation prefix (for example, select;, insert;, update;, delete; - see the native procedure logic below) must be present in the native query, but it will not be issued as part of the query to the source.

The following is an example DDL for an LDAP native procedure:

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS
("teiid_rel:native-query" 'search;context-name=corporate;filter=(&
(objectCategory=person)(objectClass=user)(!cn=$2));count-
limit=5;timeout=$1;search-scope=ONELEVEL_SCOPE;attributes=uid,cn') returns
(col1 string, col2 string);
```

NOTE

Parameter values have reserved characters escaped, but are otherwise directly substituted into the query.

13.18.5. LDAP Translator: Native Procedure

WARNING

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the translator property called "SupportsNativeQueries" to true. See Section 13.6, “Override Execution Properties”. above.

LDAP translator provides a procedure with name **native** that gives ability to execute any ad hoc native LDAP queries directly against the source without any JBoss Data Virtualization parsing or resolving. The metadata of this procedure's execution results are not known to JBoss Data Virtualization, and they are returned as object array. Users can use the ARRAYTABLE construct (Section 3.6.9, “Nested Tables: ARRAYTABLE”) to build tabular output for consumption by client applications. Since there is no known direct query language for LDAP, JBoss Data Virtualization exposes this procedure with a simple query structure as below.

13.18.6. LDAP Translator Example: Search

Example 13.7. Search Example

```
SELECT x.* FROM (call pm1.native('search;context-name=corporate;filter=
(objectClass=*);count-limit=5;timeout=6;search-
scope=ONELEVEL_SCOPE;attributes=uid,cn')) w,
 ARRAYTABLE(w.tuple COLUMNS "uid" string , "cn" string) AS x
```

The **"search"** keyword is followed by the below properties. Each property must be delimited by semicolon (;) If a property contains a semicolon (;), it must be escaped by another semicolon. See also Section 13.7, “Parameterizable Native Queries” and the example in Section 13.18.4, “LDAP Translator: Native Queries”.

Name	Description	Required
context-name	LDAP Context name	Yes
filter	query to filter the records in the context	No
count-limit	limit the number of results. same as using LIMIT	No
timeout	Time out the query if not finished in given milliseconds	No
search-scope	LDAP search scope, one of SUBTREE_SCOPE, OBJECT_SCOPE, ONELEVEL_SCOPE	No

Name	Description	Required
attributes	attributes to retrieve	Yes

13.18.7. LDAP Translator Example: Delete

Example 13.8. Delete Example

```
SELECT x.* FROM (call
pm1.native('delete;uid=doe,ou=people,o=teiid.org')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

In the above code, the "**delete**" keyword is followed by the "DN" string. All the string contents after the "delete;" are used as the DN.

13.18.8. LDAP Translator Example: Create and Update

Example 13.9. Create Example

```
SELECT x.* FROM
 (call
pm1.native('create;uid=doe,ou=people,o=teiid.org;attributes=one,two,thre
e', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

In the above code, the "**create**" keyword is followed by the "DN" string. All the string contents after the "create;" is used as the DN. It also takes one property called "attributes" which is comma separated list of attributes. The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to create:

Example 13.10. Update Example

```
SELECT x.* FROM
 (call
pm1.native('update;uid=doe,ou=people,o=teiid.org;attributes=one,two,thre
e', 'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

IMPORTANT

By default, the name of the procedure that executes the queries directly is called **native**, however this can be changed by overriding an execution property in the `vdb.xml` file. See Section 13.6, “Override Execution Properties”.

13.18.9. LDAP Connector Capabilities Support

LDAP does not provide the same set of functionality as a relational database. The LDAP Connector supports many standard SQL constructs, and performs the job of translating those constructs into an equivalent LDAP search statement. For example, the SQL statement:

```
SELECT firstname, lastname, guid
FROM public_views.people
WHERE
(lastname='Jones' and firstname IN ('Michael', 'John'))
OR
guid > 600000
```

Uses a number of SQL constructs, including:

- **SELECT** clause support
- select individual element support (firstname, lastname, guid)
- **FROM** support
- **WHERE** clause criteria support
- nested criteria support
- AND, OR support
- Compare criteria (Greater-than) support
- **IN** support

The LDAP Connector executes LDAP searches by pushing down the equivalent LDAP search filter whenever possible, based on the supported capabilities. JBoss Data Virtualization automatically provides additional database functionality when the LDAP Connector does not explicitly provide support for a given SQL construct. In these cases, the SQL construct cannot be pushed down to the data source, so it will be evaluated in JBoss Data Virtualization, in order to ensure that the operation is performed.

In cases where certain SQL capabilities cannot be pushed down to LDAP, JBoss Data Virtualization pushes down the capabilities that are supported, and fetches a set of data from LDAP. JBoss Data Virtualization then evaluates the additional capabilities, creating a subset of the original data set. Finally, JBoss Data Virtualization will pass the result to the client. It is useful to be aware of unsupported capabilities, in order to avoid fetching large data sets from LDAP when possible.

13.18.10. LDAP Connector Capabilities Support List

The following capabilities are supported in the LDAP Connector, and will be evaluated by LDAP:

- **SELECT** queries
- **SELECT** element pushdown (for example, individual attribute selection)
- **AND** criteria
- Compare criteria (e.g. <, <=, >, >=, =, !=)

- **IN** criteria
- **LIKE** criteria.
- **OR** criteria
- **INSERT**, **UPDATE**, **DELETE** statements (must meet Modeling requirements)

Due to the nature of the LDAP source, the following capability is not supported:

- **SELECT** queries

The following capabilities are not supported in the LDAP Connector, and will be evaluated by the JBoss Data Virtualization after data is fetched by the connector:

- Functions
- Aggregates
- **BETWEEN** Criteria
- Case Expressions
- Aliased Groups
- Correlated Subqueries
- **EXISTS** Criteria
- Joins
- Inline views
- **IS NULL** criteria
- **NOT** criteria
- **ORDER BY**
- Quantified compare criteria
- Row Offset
- Searched Case Expressions
- Select Distinct
- Select Literals
- **UNION**
- XA Transactions

The ldap-as-a-datasource quickstart demonstrates use of the ldap Translator to access data in the OpenLDAP Server. The name of the translator to use in vdb.xml is "translator-ldap"

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<vdb name="ldapVDB" version="1">
<model name="HRModel">
<source name="local" translator-name="translator-ldap" connection-jndi-
name="java:/ldapDS"/>
</model>
</vdb>
```

The translator does not provide a connection to the OpenLDAP. For that purpose, Teiid has a JCA adapter that provides a connection to OpenLDAP using the Java Naming API. To define such connector, use the following XML fragment in standalone-teiid.xml. See a example in "JBOSS-HOME/docs/teiid/datasources/ldap"

```
<resource-adapter id="ldapQS">
<module slot="main" id="org.jboss.teiid.resource-adapter.ldap"/>
<connection-definitions>
<connection-definition class-
name="org.teiid.resource.adapter.ldap.LDAPManagedConnectionFactory" jndi-
name="java:/ldapDS" enabled="true" use-java-context="true" pool-
name="ldapDS">
<config-property name="LdapAdminUserPassword">
redhat
</config-property>
<config-property name="LdapAdminUserDN">
cn=Manager,dc=example,dc=com
</config-property>
<config-property name="LdapUrl">
ldap://localhost:389
</config-property>
</connection-definition>
</connection-definitions>
</resource-adapter>
```

The code above defines the translator and connector. For more ways to create the connector see LDAP Data Sources, LDAP translator can derive the metadata based on existing Users/Groups in LDAP Server, user need to define the metadata. For example, you can define a schema using DDL:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="ldapVDB" version="1">
<model name="HRModel">
<metadata type="DDL"><![CDATA[
CREATE FOREIGN TABLE HR_Group (
DN string options (nameinsource 'dn'),
SN string options (nameinsource 'sn'),
UID string options (nameinsource 'uid'),
MAIL string options (nameinsource 'mail'),
NAME string options (nameinsource 'cn')
) OPTIONS(nameinsource 'ou=HR,dc=example,dc=com', updatable true);
</metadata>
</model>
</vdb>
```

When the SELECT operation below eis xecuted against a table using Teiid, it will retrieve the Users/Groups from the LDAP Server:

```
SELECT * FROM HR_Group
```

13.18.11. LDAP Attribute Datatype Support

LDAP providers currently return attribute value types of **java.lang.String** and **byte[]**, and do not support the ability to return any other attribute value type. The LDAP Connector currently supports attribute value types of **java.lang.String** only. Therefore, all attributes are modeled using the String datatype in Teiid Designer.

Conversion functions that are available in JBoss Data Virtualization allow you to use models that convert a String value from LDAP into a different data type. Some conversions may be applied implicitly, and do not require the use of any conversion functions. Other conversions must be applied explicitly, via the use of **CONVERT** functions.

Since the **CONVERT** functions are not supported by the underlying LDAP system, they will be evaluated in JBoss Data Virtualization. Therefore, if any criteria is evaluated against a converted datatype, that evaluation cannot be pushed to the data source, since the native type is String.

NOTE

When converting from String to other types, be aware that criteria against that new data type will not be pushed down to the LDAP data source. This may decrease performance for certain queries.

As an alternative, the data type can remain a string and the client application can make the conversion, or the client application can circumvent any LDAP supports <= and >=, but has no equivalent for < or >. In order to support < or > pushdown to the source, the LDAP Connector will translate < to <=, and it will translate > to >=.

When using the LDAP Connector, be aware that strictly-less-than and strictly-greater-than comparisons will behave differently than expected. It is advisable to use <= and >= for queries against an LDAP based data source, since this has a direct mapping to comparison operators in LDAP.

13.18.12. LDAP: Testing Your Connector

You must define LDAP Connector properties accurately or the JBoss Data Virtualization server will return unexpected results, or none at all. As you deploy the connector in Console, improper configuration can lead to problems when you attempt to start your connector. You can test your LDAP Connector in Teiid Designer prior to Console deployment by submitting queries at modeling time for verification.

13.18.13. LDAP: Console Deployment Issues

The Console shows an Exception That Says Error Synchronizing the Server

If you receive an exception when you synchronize the server and your LDAP Connector is the only service that does not start, it means that there was a problem starting the connector. Verify whether you have correctly typed in your connector properties to resolve this issue.

13.19. LOOPBACK TRANSLATOR

The Loopback translator, known by the type name loopback, provides a quick testing solution. It supports all SQL constructs and returns default results, with some configurable behaviour.

Table 13.11. Registry Properties

Name	Description	Default
`ThrowError`	True to always throw an error	false
`RowCount`	Rows returned for non-update queries.	1
`WaitTime`	True to always throw an error	false
`PollIntervalInMilli`	if positive results will be "asynchronously" returned - that is a DataNotAvailableException will be thrown initially and the engine will wait the poll interval before polling for the results.	-1
`DelegateName`	Set to the name of the translator which is to be mimicked.	-

You can also use the Loopback translator to mimic how a real source query would be formed for a given translator (although loopback will still return dummy data that may not be useful for your situation). To enable this behavior, set the DelegateName property to the name of the translator you wish to mimic. For example to disable all capabilities, set the DelegateName property to "jdbc-simple".

A source connection is not required for this translator.

13.20. MICROSOFT EXCEL TRANSLATOR

The Microsoft Excel Translator, known by the type name excel, exposes querying functionality to Excel documents using File Data Sources. Microsoft Excel is a popular spreadsheet software that is used by all the organizations across the globe for simple reporting purposes. This translator provides an easy way read a Excel spreadsheet and provide contents of the spreadsheet in the tabular form that can be integrated with other sources in Teiid.

NOTE

Note that this translator works on all platforms, including Windows and Linux. This translator uses Apache POI libraries to access the Excel documents which are platform independent.

This table describes how Excel translator interprets the data in Excel document into relational terms:

Table 13.12. Translation

Excel Term	Relational Term
Workbook	schema
Sheet	Table
Row	Row of data
Cell	Column Definition or Data of a column

Excel translator supports "source metadata" feature, where given Excel workbook, it can introspect and build the schema based on the Sheets defined inside it. There are options available to detect header columns and data columns in a work sheet to define the correct metadata of a table.

Here is an example of Dynamic VDB, that shows you how to expose an Excel spreadsheet:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="excelvdb" version="1">
    <model name="excel">
        <property name="importer.headerRowNumber" value="1"/>
        <property name="importer.ExcelFileName" value="names.xls"/>
        <source name="connector" translator-name="excel"  connection-jndi-
name="java:/fileDS"/>
    </model>
</vdb>
```

"connection-jndi-name" in the code sample above represents the connection to the Excel document. The Excel translator does NOT provide a connection to the Excel Document. For that purpose, Teiid uses File JCA adapter that provides a connection to Excel. To define such connector, see File Data Sources or see an example in **`jboss-as/docs/teiid/datasources/file`**. Once you configure both of the above, you can deploy them to Teiid Server and access the Excel Document using either the JDB, ODBC or OData protocol.

If you are using Designer Tooling, to create Excel based VDB, use a Teiid Designer Model project. Use "Teiid Connection - Source Model" importer, create File Data Source using data source creation wizard and use excel as translator in the importer. Based on the Excel document relevant relational tables will be created. Create a VDB and deploy into Teiid Server and and access the Excel Document using JDBC/ODBC/OData protocol.

NOTE

If you have headers in the Excel document, you can guide the import process to select the cell headers as the column names in the table creation process. See "Import Properties" section below on defining the "import" properties.

Import properties guide the schema generation part during the deployment of the VDB. This can be used in Dynamic VDBs or while using "Teiid Connection >> Source Model" in Teiid Designer.

Table 13.13. Import Properties

Property	Description	Default
importer.excelFileName	Defines the name of the Excel Document	required
importer.headerRowNumber	optional, default is first data row of sheet	required
importer.dataRowNumber	optional, default is first data row of sheet	required

NOTE

Red Hat recommend that you define all the above importer properties, so that information inside the Excel document is correctly interpreted.

Currently there are no Translator Extension properties defined for this translator.

Metadata Extension Properties are the properties that are defined on the schema artifacts like Table, Column, Procedure to describe how the translator interacts with source systems. All the properties are defined with namespace "{http://www.teiid.org/translator/excel/2014\}", which also has a recognized alias "teiid_excel".

Table 13.14. Metadata Extension Properties

Property	Schema Item	Description	Mandatory?
FILE	Table	Defines Excel Document name or name pattern	yes
FIRST_DATA_ROW_NUMBER	Table	Defines the row number where records start	Optional
CELL_NUMBER	Column of Table	Defines cell number to use for reading data of particular column	Yes

Here is an example table that is defined using the Extension Metadata Properties:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="excelvdb" version="1">
    <model name="excel">
        <source name="connector" translator-name="excel"  connection-jndi-
name="java:/fileDS"/>
         <metadata type="DDL"><![CDATA[
             CREATE FOREIGN TABLE Person (
                ROW_ID integer OPTIONS (SEARCHABLE 'All_Except_Like',
"teiid_excel:CELL_NUMBER" 'ROW_ID'),
                FirstName string OPTIONS (SEARCHABLE 'Unsearchable',
"teiid_excel:CELL_NUMBER" '1'),
                LastName string OPTIONS (SEARCHABLE 'Unsearchable',
"teiid_excel:CELL_NUMBER" '2'),
                Age integer OPTIONS (SEARCHABLE 'Unsearchable',
"teiid_excel:CELL_NUMBER" '3'),
                CONSTRAINT PK0 PRIMARY KEY(ROW_ID)
             ) OPTIONS ("NAMEINSOURCE" 'Sheet1',"teiid_excel:FILE"
'names.xlsx', "teiid_excel:FIRST_DATA_ROW_NUMBER" '2')
        ]> </metadata>
    </model>
</vdb>
```

NOTE

"Extended capabilities using ROW_ID column" If you define column, that has extension metadata property "CELL_NUMBER" with value "ROW_ID", then that column value contains the row information from Excel document. You can mark this column as Primary Key. You can use this column in SELECT statements with a restrictive set of capabilities including: comparison predicates, IN predicates and LIMIT. All other columns can not be used as predicates in a query.

NOTE

The user does not have to depend upon "source metadata" import, or Designer tool import to create the schema represented by Excel document, they can manually create a source table and add the appropriate extension properties to make a fully functional model. If you introspect the schema model created by the import, it would look like the code above.

There is no Teiid-specific Excel Resource Adapter. Use the File JCA adapter with this translator.

The Excel translator does not yet support updates.

13.21. MONGODB TRANSLATOR

13.21.1. MongoDB

MongoDB is a document based "schema-less" database with it own query language. It does not map perfectly with relational concepts or the SQL query language. More and more systems are using this type of NoSQL store for scalability and performance. For example, applications like storing audit logs or managing web site data fits well with MongoDB, and does not require using a structural database like Oracle, Postgres ect. MongoDB uses JSON documents as its primary storage unit, and it can have additional embedded documents inside the parent document. By using embedded documents it co-locates the related information to achieve de-normalization that typically requires either duplicate data or joins to achieve in a relational database.

For MongoDB to work with JBoss Data Virtualization, the challenge for the MongoDB translator is to design a MongoDB store that can achieve the balance between relational and document based storage. In our opinion the advantages of "schema-less" design are great at development time. "Schema-less" can also be a problem with migration of application versions and the ability to query and make use of returned information effectively.

Since it is hard and may be impossible in certain situations to derive a schema based on existing the MongoDB collection(s), JBoss Data Virtualization approaches the problem in reverse compared to other translators. When working with MongoDB, JBoss Data Virtualization requires the user to define the MongoDB schema upfront using JBoss Data Virtualization metadata. Since JBoss Data Virtualization only allows relational schema as its metadata, the user needs to define their MongoDB schema in relational terms using tables, procedures, and functions. For the purposes of MongoDB, the JBoss Data Virtualization metadata has been extended to support extension properties that can be defined on the table to convert it into a MongoDB based document. These extension properties let users define how a MongoDB document is structured and stored. Based on the relationships (primary-key, foreign-key) defined on a table and the cardinality (ONE-to-ONE, ONE-to-MANY, MANY-to-ONE), relations between tables are mapped such that related information can be embedded along with the parent document for co-location (see the de-normalization comment above). Thus a relational schema based design, but document based storage in MongoDB.

13.21.2. MongoDB Translator

The MongoDB translator, known by the type name *mongodb*, provides a relational view of data that resides in a MongoDB database. This translator is capable of converting JBoss Data Virtualization SQL queries into MongoDB based queries. It supports a full range of SELECT, INSERT, UPDATE and DELETE calls.

The document structure in MongoDB can be more complex than what JBoss Data Virtualization can currently define. This translator is currently designed for:

- Users that are using relational databases and would like to move/migrate their data to MongoDB to take advantage of scaling and performance, without modifying end user applications that are currently running.
- Users that are starting out with MongoDB and do not have experience with MongoDB, but are seasoned SQL developers. This provides a low barrier of entry compared to using MongoDB directly as an application developer.
- Integrating other enterprise data sources with MongoDB based data.

NOTE

The MongoDB translator does not currently support native queries.

NOTE

The resource adapter for this translator is provided by configuring the "mongodb" data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration Guide* for more configuration information. An example configuration file is found at ***EAP_HOME*/docs/teiid/datasources/mongodb**.

13.21.3. MongoDB Translator: Example DDL

The name of the translator to use in vdb.xml is "mongodb":

```
<vdb name="nothwind" version="1">
    <model name="northwind">
        <source name="local" translator-name="mongodb" connection-jndi-
name="java:/mongoDS"/>
    </model>
<vdb>
```

The translator does not provide a connection to the MongoDB. For that purpose, Teiid has a JCA adapter that provides a connection to MongoDB using the MongoDB Java Driver. To define such connector, use the following XML fragment in standalone-teiid.xml.

```
<resource-adapters>
    <resource-adapter id="mongodb">
        <module slot="main" id="org.jboss.teiid.resource-
adapter.mongodb"/>
        <transaction-support>NoTransaction</transaction-support>
        <connection-definitions>
            <connection-definition class-
name="org.teiid.resource.adapter.mongodb.MongoDBManagedConnectionFactory"
                    jndi-name="java:/mongoDS"
                    enabled="true"
                    use-java-context="true"
                    pool-name="teiid-mongodb-ds">

                  <!-- MongoDB server list (host:port[;host:port...]) -->
                  <config-property
name="RemoteServerList">localhost:27017</config-property>
                  <!-- Database Name in the MongoDB -->
                  <config-property name="Database">test</config-property>
                    <!--
                        Uncomment these properties to supply user name
and password
                    <config-property name="Username">user</config-
property>
                    <config-property name="Password">user</config-
property>
                    -->
            </connection-definition>
        </connection-definitions>
    </resource-adapter>
</resource-adapters>
```

MongoDB translator can derive the metadata based on existing document collections in some scenarios, however when working with complex documents the interpretation of metadata may be accurate, in those situations the user MUST define the metadata. For example, you can define a schema using DDL:

```
<vdb name="nothwind" version="1">
    <model name="northwind">
        <source name="local" translator-name="mongodb" connection-jndi-
name="java:/mongoDS"/>
            <metadata type="DDL"><![CDATA[
                CREATE FOREIGN TABLE  Customer (
                    customer_id integer,
                    FirstName varchar(25),
                    LastName varchar(25)
                ) OPTIONS(UPDATABLE 'TRUE');
            ]]> </metadata>
    </model>
<vdb>
```

When this INSERT operation is executed against table using Teiid, MongoDB translator will create a document in the MongoDB.

```
INSERT INTO Customer(customer_id, FirstName, LastName) VALUES (1, 'John',
'Doe');
```

```
{
  _id: ObjectID("509a8fb2f3f4948bd2f983a0"),
  customer_id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If a PRIMARY KEY is defined on the table, then that column name is automatically used as "_id" field in the MongoDB collection, then document structure is stored in the MongoDB.

```
CREATE FOREIGN TABLE  Customer (
    customer_id integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');
```

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}
```

If you defined the composite PRIMARY KEY on Customer table, the document structure will look like this:

```
CREATE FOREIGN TABLE  Customer (
    customer_id integer,
    FirstName varchar(25),
    LastName varchar(25),
    PRIMARY KEY (FirstName, LastName)
) OPTIONS(UPDATABLE 'TRUE');
```

```
{
  _id: {
         FirstName: "John",
         LastName:  "Doe"
       },
  customer_id: 1,
}
```

MongoDB translator supports automatic mapping of Teiid data types into MongoDB data types, including the support for Blobs, Clobs and XML. The LOB support is based on GridFS in MongoDB. Arrays are in this form:

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
  Score: [89, "ninety", 91.0]
}
```

User can get individual items in the array using function array_get, or can transform the array into tabular structure using ARRATTABLE.

NOTE

Note that even though embedded documents can also be in arrays, the handling of embedded documents is different from array with scalar values.

NOTE

Regular Expressions, MongoDB::Code, MongoDB::MinKey, MongoDB::MaxKey and MongoDB::OID are not supported.

13.21.4. MongoDB Translator: Metadata Extensions

Using the above DDL or any other metadata facility, a user can map a table in a relational store into a document in MongoDB, however to make effective use of MongoDB, you need to be able to build complex documents, that can co-locate related information, so that data can queried in a single MongoDB query. Otherwise, since MongoDB does not support join relationships like relational database, you need to issue multiple queries to retrieve and join data manually. The power of MongoDB comes from its "embedded" documents and its support of complex data types like arrays and use of the aggregation framework to be able to query them. This translator provides way to achieve that goals.

When you do not define the complex embedded documents in MongoDB, Teiid can step in for join processing and provide that functionality, however if you want to make use of the power of MongoDB itself in querying the data and avoid bringing the unnecessary data and improve performance, you need to look into building these complex documents.

MongoDB translator defines two additional metadata properties along with other Teiid metadata properties to aid in building the complex "embedded" documents. You can use the following metadata properties in your DDL.

- teiid_mongo:EMBEDDABLE - Means that data defined in this table is allowed to be included as an "embeddable" document in any parent document. The parent document is referenced by the foreign key relationships. In this scenario, Teiid maintains more than one copy of the data in MongoDB store, one in its own collection and also a copy in each of the parent tables that have relationship to this table. You can even nest embeddable table inside another embeddable table with some limitations. Use this property on table, where table can exist, encompass all its relations on its own. For example, a "Category" table that defines a "Product"'s category is independent of Product, which can be embeddable in "Products" table.

- teiid_mongo:MERGE - Means that data of this table is merged with the defined parent table. There is only a single copy of the data that is embedded in the parent document. Parent document is defined using the foreign key relationships.

IMPORTANT

A given table can contain either the "teiid_mongo:EMBEDDABLE" property or the "teiid_mongo:MERGE" property defining the type of nesting in MongoDB. A table is not allowed to have both properties.

- *EMBEDDABLE* - Means that data defined in this table is allowed to be included as an "embeddable" document in a parent document. The parent document is defined by the foreign key relationships. In this situation, JBoss Data Services maintains more than one copy of the data in a MongoDB store: one in its own collection and also a copy in each of the parent tables that have relationship to this table.

- *EMBEDIN* - Means that data of this table is embedded in the defined parent table. There is only a single copy of the data that is embedded in the parent document.

These properties behave differently for particular relationship types on the schema:

- ONE-2-ONE: Here is the DDL structure representing the ONE-2-ONE relationship:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Address (
    CustomerId integer,
    Street varchar(50),
    City varchar(25),
    State varchar(25),
    Zipcode varchar(6),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
 ) OPTIONS(UPDATABLE 'TRUE');
```

  By default, this will produce two different collections in MongoDB, like with sample data it will look like this:

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Address
{
  _id: ObjectID("..."),
   CustomerId: 1,
   Street: "123 Lane"
   City: "New York",
   State: "NY"
   Zipcode: "12345"
}
```

You can enhance the storage in MongoDB to a single collection by using "teiid_mongo:MERGE' extension property on the table's OPTIONS clause:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Address (
    CustomerId integer PRIMARY KEY,
    Street varchar(50),
    City varchar(25),
    State varchar(25),
    Zipcode varchar(6),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
 ) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

This will produce a single collection in the MongoDB:

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Address:
     {
        Street: "123 Lane",
        City: "New York",
        State: "NY",
        Zipcode: "12345"
     }
}
```

Both tables are merged into a single collection that can be queried together using the JOIN clause in the SQL command. Since the existence of child/additional record has no meaning with out parent table using the "teiid_mongo:MERGE" extension property is right choice in this situation.

NOTE

Note that the Foreign Key defined on child table, must refer to Primary Keys on both parent and child tables to form a One-2-One relationship.

- ONE-2-MANY: Typically there are only two tables involved in this relationship. If MANY side is only associated one table, then use "EMBEDIN" property on MANY side of table and define the parent. If associated with more than single table, then use "EMBEDDABLE". When MANY side is stored in ONE side, they are stored as array of embedded document. If associated with more than single table then use "teiid_mongo:EMBEDDABLE".

  Here is a sample DDL:

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');
```

In this sample, a single Customer can have many orders. There are two options to define the how we store the MongoDB document. If in your schema, the Customer table's CustomerId is only referenced in Order table (i.e. Customer information used for only Order purposes), you can use

```
CREATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Customer');
```

This will produce a single document for the customer table:

```
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe",
  Order:
  [
     {
       _id: 100,
        OrderDate: ISODate("2000-01-01T06:00:00Z")
        Status: 2
     },
     {
       _id: 101,
        OrderDate: ISODate("2001-03-06T06:00:00Z")
        Status: 5
     }
     ...
    ]
}
```

If the customer table is referenced in more tables other than Order table, then use the "teiid_mongo:EMBEDDABLE" property:

```
CREATE FOREIGN TABLE Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:EMBEDDABLE" 'TRUE');

CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Comments (
    CommentID integer PRIMARY KEY,
    CustomerId integer,
    Comment varchar(140),
    FOREIGN KEY (CustomerId) REFERENCES Customer (CustomerId)
) OPTIONS(UPDATABLE 'TRUE');
```

This creates three different collections in MongoDB:

```
Customer
{
  _id: 1,
  FirstName: "John",
  LastName: "Doe"
}

Order
{
  _id: 100,
  CustomerId: 1,
  OrderDate: ISODate("2000-01-01T06:00:00Z")
  Status: 2
  Customer:
   {
     FirstName: "John",
     LastName: "Doe"
   }
}

Comment
{
  _id: 12,
  CustomerId: 1,
  Comment: "This works!!!"
  Customer:
   {
     FirstName: "John",
     LastName: "Doe"
   }
}
```

Here the Customer table contents are embedded along with other table's data where they were referenced. This creates duplicated data where multiple of these embedded documents are managed automatically in the MongoDB translator.

WARNING

All the SELECT, INSERT, DELETE operations that are generated against the tables with "teiid_mongo:EMBEDDABLE" property are atomic, except for UPDATES, as there can be multiple operations involved to update all the copies.

- MANY-2-ONE: This is the same as ONE-2-MANY. Apply them in reverse.

NOTE

A parent table can have multiple "embedded" and as well as "merge" documents inside it, it not limited so either one or other. However, please note that MongoDB imposes document size is limited can not exceed 16MB.

- Many-to-Many: This can also mapped with combination of "teiid_mongo:MERGE" and "teiid_mongo:EMBEDDABLE" properties (partially). Here is a sample DDL:

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE');
```

Modify the DDL so that it looks like this:

```
CREATE FOREIGN TABLE Order (
    OrderID integer PRIMARY KEY,
    OrderDate date,
    Status integer
) OPTIONS(UPDATABLE 'TRUE');
```

```
CREATE FOREIGN TABLE OrderDetail (
    OrderID integer,
    ProductID integer,
    PRIMARY KEY (OrderID,ProductID),
    FOREIGN KEY (OrderID) REFERENCES Order (OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product (ProductID)
) OPTIONS(UPDATABLE 'TRUE', "teiid_mongo:MERGE" 'Order');

CREATE FOREIGN TABLE Products (
   ProductID integer PRIMARY KEY,
   ProductName varchar(40)
) OPTIONS(UPDATABLE 'TRUE',  "teiid_mongo:EMBEDDABLE" 'TRUE');
```

A document that looks like this is produced:

```
{
   _id : 10248,
   OrderDate : ISODate("1996-07-04T05:00:00Z"),
   Status : 5
   OrderDetails : [
     {
       _id : {
               OrderID : 10248,
               ProductID : 11
               Products : {
                  ProductID: 11
                  ProductName: "Hammer"
               }
       }
     },
     {
       _id : {
         OrderID : 10248,
         ProductID : 14
         Products : {
             ProductID: 14
             ProductName: "Screw Driver"
         }
       }
     }
   ]
}

Products
{
    {
      ProductID: 11
      ProductName: "Hammer"
    }
    {
      ProductID: 14
      ProductName: "Screw Driver"
    }
}
```

WARNING

- Currently nested embedding of documents has limited support due to capabilities of handling nested arrays is limited in the MongoDB. Nesting of "EMBEDDABLE" property with multiple levels is allowed but more than one level with MERGE is not. Also, be careful not to exceed the document size of 16 MB for a single row, (hence deep nesting is not recommended).

- JOINS between related tables, must use either the "EMBEDDABLE" or "MERGE" property, otherwise the query will result in error. In order for Teiid to correctly plan and support the JOINS, in the case that any two tables are NOT embedded in each other, use allow-joins=false property on the Foreign Key that represents the relation. Here is an example:

```
REATE FOREIGN TABLE  Customer (
    CustomerId integer PRIMARY KEY,
    FirstName varchar(25),
    LastName varchar(25)
) OPTIONS(UPDATABLE 'TRUE');

CREATE FOREIGN TABLE  Order (
    OrderID integer PRIMARY KEY,
    CustomerId integer,
    OrderDate date,
    Status integer,
    FOREIGN KEY (CustomerId) REFERENCES Customer
(CustomerId) OPTIONS (allow-join 'FALSE')
) OPTIONS(UPDATABLE 'TRUE');
```

  In this case, Teiid will create two collections. However when a user issues query such as this, instead of resulting in error, the JOIN processing will happen in the Teiid engine, without the above property it will result in an error:

```
SELECT OrderID, LastName FROM Order JOIN Customer
ON Order.CustomerId = Customer.CustomerId;
```

MongoDB translator designed on top of the MongoDB aggregation framework, use of MongoDB version that supports this framework is mandatory. Apart from SELECT queries, this translator also supports INSERT, UPDATE and DELETE queries. It also supports grouping, matching, sorting, filtering, limit, support for LOBs using GridFS and composite primary and foreign keys.

MongoDB source procedures may be created using the teiid_rel:native-query extension. The procedure will invoke the native-query similar to a direct procedure call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE or similar functionality.

WARNING

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the execution property called SupportsDirectQueryProcedure to true.

NOTE

By default the name of the procedure that executes the queries directly is called native. Override the execution property DirectQueryProcedureName to change it to another name.

The MongoDB translator provides a procedure to execute any ad-hoc aggregate query directly against the source without Teiid parsing or resolving. Since the metadata of this procedure's results are not known to Teiid, they are returned as an object array containing single blob at array location one(1). This blob contains the JSON document. XMLTABLE can be used construct tabular output for consumption by client applications.

```
select x.* from TABLE(call native('city;{$match:{"city":"FREEDOM"}}')) t,
      xmltable('/city' PASSING JSONTOXML('city', cast(array_get(t.tuple,
1) as BLOB)) COLUMNS city string, state string) x
```

In this example, a collection called "city" is looked up with filter that matches the "city" name with "FREEDOM", using "native" procedure and then using the nested tables feature the output is passed to a XMLTABLE construct, where the output from the procedure is sent to a JSONTOXML function to construct a XML then the results of that are exposed in tabular form.

IMPORTANT

The direct query must be in this format:

```
"collectionName;{$pipeline instr}+"
```

MongoDB translator also allows to execute Shell type java script commands like remove, drop, createIndex.

The commands need to be in this format:

```
"$ShellCmd;collectionName;operationName;{$instr}+"
```

Here is an example:

```
"$ShellCmd;MyTable;remove;{ qty: { $gt: 20 }}"
```

13.22. OBJECT TRANSLATOR

13.22.1. Object Translator

The Object translator is a bridge for reading Java objects from external sources, such as JBoss Data Grid (**`infinispan-cache`**) or Map Cache, and delivering them to the engine for processing. To assist in providing that bridge, the OBJECTTABLE function (Section 3.6.10, "Nested Tables: OBJECTTABLE") must be used to transform the Java object into rows and columns.

The following are Object translator types:

- **`map-cache`** - supports a local cache that is of type Map and using Key searching. This translator is implemented by the **`org.teiid.translator.object.ObjectExecutionFactory`** class.
- **`infinispan-cache`** - supports JBoss Data Grid using either Key searching (for objects that are not annotated) or Hibernate/Lucene searching. This translator is implemented by the **`org.teiid.translator.object.infinispan.InfinispanExecutionFactory`** class which extends the **`org.teiid.translator.object.ObjectExecutionFactory`** class.

NOTE

See the JBoss Data Grid resource adapter for this translator. It can be configured to lookup the cache container via JNDI or created (i.e., ConfigurationFileName or RemoteServerList). Also see the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.22.2. Object Translator: Execution Properties

The following execution properties are relevant to translating from JBoss Data Grid.

Table 13.15. Execution Properties

Name	Description	Required	Default
SupportsLuceneSearching	Setting to true assumes your objects are annotated and Hibernate/Lucene will be used to search the cache	N	false

13.22.3. Object Translator: Supported Capabilities

The following are the connector capabilities when Key Searching is used:

- SELECT command
- CompareCriteria - only EQ
- InCriteria

The following are the connector capabilities when Hibernate/Lucene Searching is enabled:

- SELECT command

- CompareCriteria - EQ, NE, LT, GT, etc.
- InCriteria
- OrCriteria
- And/Or Criteria
- Like Criteria
- INSERT, UPDATE, DELETE

13.22.4. Object Translator: Usage

Retrieve objects from a cache and transform into rows and columns.

- The primary object returned by the cache should have a name in source of 'this'. All other columns will have their name in source (which defaults to the column name) interpreted as the path to the column value from the primary object.
- All columns that are not the primary key nor covered by a lucene index should be marked as SEARCHABLE 'Unsearchable'.

13.22.5. Object Translator Example

The following is an example of a key search. It uses a dynamic vdb to define the physical source and views using DDL. It uses a TeamObject class, shown below, with a teamName field that is used as its cache key and a String list of players.

```
public class TeamObject {

   private String teamName;
   private List<String> players = new ArrayList<String>();

   public String getTeamName() {
      return teamName;
   }

   public void setTeamName(String teamName) {
           this.teamName = teamName;
   }

   public List<String> getPlayers() {
           return players;
   }

}
```

Notice the use of the [OBJECTABLE|TEIID:OBJECTABLE] function to parse the object from Team and transform into rows and column. This metadata could also be defined by using Teiid Designer.

```
<vdb name="team" version="1">
    <property name="UseConnectorMetadata" value="cached" />
    <model name="Team" visible="false">
        <source name="objsource" translator-name="infinispan1" connection-
```

```
jndi-name="java:infinispan-jndi"/>
        <metadata type="DDL"><![CDATA[

            CREATE FOREIGN TABLE Team (
                TeamObject Object OPTIONS (NAMEINSOURCE 'this',
SEARCHABLE 'Unsearchable'),
                teamName varchar(255) PRIMARY KEY)
              OPTIONS (NAMEINSOURCE 'teams');

         ]]> </metadata>
    </model>
    <model name="TeamView" type="VIRTUAL">
         <metadata type="DDL"><![CDATA[
             CREATE VIEW Players (
                  TeamName varchar(255) PRIMARY KEY,
                  PlayerName varchar(255)
             )
             AS
             SELECT t.TeamName, y.Name FROM Team as T,
                   OBJECTTABLE('m.players' PASSING T.TeamObject as m
COLUMNS Name string 'teiid_row') as y;

        ]]> </metadata>
    </model>

    <translator name="infinispan1" type="infinispan-cache">
        <property name="SupportsLuceneSearching" value="true"/>
    </translator>
</vdb>
```

13.23. ODATA TRANSLATOR

13.23.1. OData Translator

The OData translator exposes the OData V2 and V3 data sources. This translator implements a simple connection for web services in the same way as the Web Services translator.

The OData translator is implemented by the **`org.teiid.translator.odata.ODataExecutionFactory`** class and known by the translator type name **`odata`**.

NOTE

Open Data Protocol (OData) is a Web protocol for querying and updating data that provides a way to unlock your data and free it from silos that exist in applications today. OData does this by applying and building upon Web technologies such as HTTP, Atom Publishing Protocol (AtomPub) and JSON to provide access to information from a variety of applications, services, and stores. OData is being used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems and traditional Web sites.

Using this specification from OASIS group, and with the help from framework OData4J, JBoss Data Virtualization maps OData entities into relational schema. JBoss Data Virtualization supports reading of CSDL (Conceptual Schema Definition Language) from the OData endpoint provided and converts the

OData schema into relational schema. The below table shows the mapping selections in OData Translator from CSDL document.

OData	Mapped to Relational Entity
EntitySet	Table
FunctionImport	Procedure
AssosiationSet	Foreign Keys on the Table*
ComplexType	ignored**

* A Many to Many association will result in a link table that can not be selected from, but can be used for join purposes.

** When used in Functions, an implicit table is exposed. When used to define a embedded table, all the columns will be in-lined.

All CRUD operations will be appropriately mapped to the resulting entity based on the SQL submitted to the OData translator.

NOTE

The resource adapter for this translator is provided by configuring the `webservice` data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.23.2. OData Translator: Execution Properties

Table 13.16. Execution Properties

Name	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetching date, time, or timestamp values	The system default time zone

13.23.3. OData Translator: Importer Properties

Table 13.17. Importer Properties

Name	Description	Default
schemaNamespace	Namespace of the schema to import	null
entityContainer	Entity Container Name to import	default container

Example importer settings to only import tables and views from NetflixCatalog:

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.schemaPattern" value="NetflixCatalog"/>
```

13.23.4. OData Translator: Usage

Usage of an OData source is similar to a JDBC translator. The metadata import is supported through the translator, once the metadata is imported from source system and exposed in relational terms, then this source can be queried as if the EntitySets and Function Imports were local to the JBoss Data Virtualization system.

Table 13.18. Execution Properties

Property	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values	The system default time zone
SupportsOdataCount	Supports $count	True
SupportsOdataFilter	Supports $filter	True
SupportsOdataOrderBy	Supports $orderby	true
SupportsOdataSkip	Supports $skip	True
SupportsOdataTop	Supports $top	True

Table 13.19. Importer Properties

Property	Description	Default
schemaNamespace	Namespace of the schema to import	Null
entityContainer	Entity Container Name to import	Default container

Here are some importer settings to import tables and views only from NetflixCatalog:

```
<property name="importer.schemaNamespace" value="System.Data.Objects"/>
<property name="importer.schemaPattern" value="NetflixCatalog"/>
```

NOTE

Sometimes it's possible that the odata server you are querying does not fully implement all OData specification features. If your OData implementation does not support a certain feature, then turn off the corresponding capability using "execution Properties", so that Teiid will not pushdown invalid queries to the translator. For example, to turn off $filter you add following to your vdb.xml then use "odata-override" as the translator name on your source model:

```
<translator name="odata-override" type="odata">
<property name="SupportsOdataFilter" value="false"/>
</translator>
```

NOTE

Native or direct query execution is not supported through OData translator. However, user can use Web Services Translator's invokehttp method directly to issue a Rest based call and parse results using SQLXML.

NOTE

Teiid can not only consume OData based data sources, but it can expose any data source as an Odata based webservice. For more information see OData Support.

13.24. OLAP TRANSLATOR

13.24.1. OLAP Translator

The OLAP translator exposes stored procedures for calling analysis services backed by an OLAP server using MDX query language.

The OLAP translator is implemented by the **`org.teiid.translator.olap.OlapExecutionFactory`** class and known by the translator type name **`olap`**.

This translator exposes a stored procedure, invokeMDX, that returns a result set containing tuple array values for a given MDX query. invokeMDX will commonly be used with the ARRAYTABLE table function (Section 3.6.9, “Nested Tables: ARRAYTABLE”) to extract the results.

Since the Cube metadata exposed by the OLAP servers and relational database metadata are so different, there is no single way to map the metadata from one to other. It is best to query OLAP system using its own native MDX language through. MDX queries my be defined statically or built dynamically in the JBoss Data Virtualization abstraction layers.

NOTE

The resource adapter for this translator is provided by configuring the data source in the JBoss EAP instance. Two sample datasource files are provided for accessing OLAP servers. One is Mondrian specific when the Mondrian server is deployed in the same JBoss EAP instance as JBoss Data Virtualization (**`mondrian.xml`**). To access any other OLAP servers using XMLA interface, the data source for them can be created using the example template in **`olap-xmla.xml`**. These example files can be found in the ***`EAP_HOME`*`/docs/teiid/datasources/`** directory. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.24.2. OLAP Translator: Usage

The OLAP translator exposes one low level procedure for accessing OLAP services: **`invokeMDX`**.

`invokeMdx` returns a result set of the tuples as array values.

```
Procedure invokeMdx(mdx in STRING, params VARIADIC OBJECT) returns table
(tuple object)
```

The mdx parameter is a MDX query to be executed on the OLAP server.

The results of the query will be returned such that each row on the row axis will be packed into an array value that will first contain each hierarchy member name on the row axis then each measure value from the column axis.

NOTE

Consider using data roles to prevent arbitrary MDX from being submitted to the invokeMDX procedure. See Section 7.1, “Data Roles”.

13.24.3. OLAP Translator: Native Queries

OLAP source procedures may be created using the teiid_rel:native-query extension. See Section 13.7, “Parameterizable Native Queries”.

NOTE

The parameter value substitution directly inserts boolean, and number values, and treats all other values as string literals.

The procedure will invoke the native query similar to an invokeMdx call with the benefits that the query is predetermined and that result column types are known, rather than requiring the use of ARRAYTABLE (Section 3.6.9, “Nested Tables: ARRAYTABLE”) or similar functionality.

13.24.4. OLAP Translator: Native Procedure

The invokeMdx procedure is the native procedure for the OLAP translator. It may be disabled or have its name changed via the common native translator properties like any other source. A call to a native procedure without any parameters will not attempt to parse the MDX query for parameterization. If parameters are used, the value substitution directly inserts boolean, and number values, and treats all other values as string literals.

13.25. SALESFORCE TRANSLATOR

13.25.1. Salesforce Translator

The Salesforce translator supports the SELECT, DELETE, INSERT and UPDATE operations against a Salesforce.com account.

The Salesforce translator is implemented by the `org.teiid.translator.salesforce.SalesForceExecutionFactory` class and known by the translator type name `salesforce`.

NOTE

The resource adapter for this translator is provided by configuring the `salesforce` data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

13.25.2. Salesforce Translator: Execution Properties

Table 13.20. Execution Properties

Name	Description	Default
ModelAuditFields	Audit Model Fields	false
MaxBulkInsertBatchSize	Batch size to use when inserting in bulk	2048

Table 13.21. Import Properties

Name	Description	Required?	Default
NormalizeNames	If the importer should attempt to modify the object/field names so that they can be used unquoted.	false	True

The Salesforce translator can import metadata.

Table 13.22. Import Properties

Name	Description	Required	Default
NormalizeNames	If the importer should attempt to modify the object/field names so that they can be used unquoted.	false	true

Name	Description	Required	Default
excludeTables	A case-insensitive regular expression that when matched against a table name will exclude it from import. Applied after table names are retrieved. Use a negative look-ahead inclusion pattern to act as an inclusion filter.	false	n/a
includeTables	A case-insensitive regular expression that when matched against a table name will be included during import. Applied after table names are retrieved from source.	false	n/a
importStatstics	Retrieves cardinalities during import using the REST API explain plan feature.	false	false

When both includeTables and excludeTables patterns are present during the import, the includeTables pattern matched first, then the excludePatterns will be applied.

13.25.3. Salesforce Translator: SQL Processing

Salesforce does not provide the same set of functionality as a relational database. For example, Salesforce does not support arbitrary joins between tables. However, working in combination with the JBoss Data Virtualization Query Planner, the Salesforce connector supports nearly all of the SQL syntax supported by JBoss Data Virtualization.

The Salesforce Connector executes SQL commands by pushing down the command to Salesforce whenever possible, based on the supported capabilities. JBoss Data Virtualization will automatically provide additional database functionality when the Salesforce Connector does not explicitly provide support for a given SQL construct. In these cases, the SQL construct cannot be pushed downto the data source, so it will be evaluated in JBoss Data Virtualization, in order to ensure that the operation is performed.

In cases where certain SQL capabilities cannot be pushed down to Salesforce, JBoss Data Virtualization will push down the capabilities that are supported, and fetch a set of data from Salesforce. Then, JBoss Data Virtualization will evaluate the additional capabilities, creating a subset of the original data set. Finally, JBoss Data Virtualization will pass the result to the client.

```
SELECT sum(Reports) FROM Supervisor where Division = 'customer support';
```

Neither Salesforce nor the Salesforce Connector support the sum() scalar function, but they do support CompareCriteriaEquals, so the query that is passed to Salesforce by the connector will be transformed to this query.

```
SELECT Reports FROM Supervisor where Division = 'customer support';
```

The sum() scalar function will be applied by the JBoss Data Virtualization Query Engine to the result set returned by the connector.

In some cases multiple calls to the Salesforce application will be made to support the SQL passed to the connector.

```
DELETE From Case WHERE Status = 'Closed';
```

The API in Salesforce to delete objects only supports deleting by ID. In order to accomplish this, the Salesforce connector will first execute a query to get the IDs of the correct objects, and then delete those objects. So the above DELETE command will result in the following two commands.

```
SELECT ID From Case WHERE Status = 'Closed';
DELETE From Case where ID IN (<result of query>);
```

The Salesforce API DELETE call is not expressed in SQL, but the above is an SQL equivalent expression.

It is useful to be aware of unsupported capabilities, in order to avoid fetching large data sets from Salesforce and making your queries perform as well as possible.

13.25.4. Salesforce Translator: Multi-Select Picklists

A multi-select pick list is a field type in Salesforce that can contain multiple values in a single field. Query criteria operators for fields of this type in Salesforce Object Query Language (SOQL) are limited to EQ, NE, includes and excludes. The full Salesforce documentation for selecting from mullti-select pick lists can be found at Querying Mulit-select Picklists.

JBoss Data Virtualization SQL does not support the includes or excludes operators, but the Salesforce connector provides user defined function definitions for these operators that provide equivalent functionality for fields of type multi-select. The definition for the functions are:

```
boolean includes(Column column, String param)
boolean excludes(Column column, String param)
```

For example, take a single multi-select picklist column called Status that contains all of these values.

- current
- working
- critical

For that column, all of the below are valid queries:

```
SELECT * FROM Issue WHERE true = includes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = excludes (Status, 'current, working' );
SELECT * FROM Issue WHERE true = includes (Status, 'current;working,
critical' );
```

EQ and NE criteria will pass to Salesforce as supplied. For example, these queries will not be modified by the connector.

```
SELECT * FROM Issue WHERE Status = 'current';
SELECT * FROM Issue WHERE Status = 'current;critical';
SELECT * FROM Issue WHERE Status != 'current;working';
```

13.25.5. Salesforce Translator: Selecting All Objects

The Salesforce connector supports calling the queryAll operation from the Salesforce API. The queryAll operation is equivalent to the query operation with the exception that it returns data about all current and deleted objects in the system.

The connector determines if it will call the query or queryAll operation via reference to the isDeleted property present on each Salesforce object, and modeled as a column on each table generated by the importer. By default this value is set to False when the model is generated and thus the connector calls query. Users are free to change the value in the model to True, changing the default behavior of the connector to be queryAll.

The behavior is different if isDeleted is used as a parameter in the query. If the isDeleted column is used as a parameter in the query, and the value is 'true' the connector will call queryAll.

```
select * from Contact where isDeleted = true;
```

If the isDeleted column is used as a parameter in the query, and the value is 'false' the connector performing the default behavior will call the query.

```
select * from Contact where isDeleted = false;
```

13.25.6. Salesforce Translator: Selecting Updated Objects

If the option is selected when importing metadata from Salesforce, a GetUpdated procedure is generated in the model with the following structure:

```
GetUpdated (ObjectName IN string,
   StartDate IN datetime,
   EndDate IN datetime,
   LatestDateCovered OUT datetime)
returns
   ID string
```

See the description of the GetUpdated operation in the Salesforce documentation for usage details.

13.25.7. Salesforce Translator: Selecting Deleted Objects

If the option is selected when importing metadata from Salesforce, a GetDeleted procedure is generated in the model with the following structure:

```
GetDeleted (ObjectName IN string,
   StartDate IN datetime,
   EndDate IN datetime,
   EarliestDateAvailable OUT datetime,
   LatestDateCovered OUT datetime)
returns
   ID string,
   DeletedDate datetime
```

–

See the description of the GetDeleted operation in the Salesforce documentation for usage details.

13.25.8. Salesforce Translator: Relationship Queries

Salesforce does not support joins like a relational database, but it does have support for queries that include parent-to-child or child-to-parent relationships between objects. These are termed Relationship Queries. The SalesForce connector supports Relationship Queries through Outer Join syntax.

```
SELECT Account.name, Contact.Name from Contact LEFT OUTER JOIN Account
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from child to parent. It resolves to the following query to SalesForce.

```
SELECT Contact.Account.Name, Contact.Name FROM Contact
```

```
select Contact.Name, Account.Name from Account Left outer Join Contact
on Contact.Accountid = Account.id
```

This query shows the correct syntax to query a SalesForce model with to produce a relationship query from parent to child. It resolves to the following query to SalesForce.

```
SELECT Account.Name, (SELECT Contact.Name FROM
Account.Contacts) FROM Account
```

See the description of the Relationship Queries operation in the SalesForce documentation for limitations.

13.25.9. Salesforce Translator: Bulk Insert Queries

SalesForce translator also supports bulk insert statements using JDBC batch semantics or SELECT INTO semantics. The batch size is determined by the execution property MaxBulkInsertBatchSize , which can be overridden in the `vdb.xml` file. The default value of the batch is 2048. The bulk insert feature uses the async REST based API exposed by Salesforce for execution for better performance.

13.25.10. Salesforce Translator: Supported Capabilities

The following are the connector capabilities supported by the Salesforce Connector. These SQL constructs will be pushed down to Salesforce.

- SELECT command
- INSERT Command
- UPDATE Command
- DELETE Command
- CompareCriteriaEquals
- InCriteria

- LikeCriteria - Supported for String fields only.
- RowLimit
- AggregatesCountStar
- NotCriteria
- OrCriteria
- CompareCriteriaOrdered
- OuterJoins with join criteria KEY

13.25.11. Salesforce Translator: Native Queries

Salesforce procedures may optionally have native queries associated with them. See Section 13.7, “Parameterizable Native Queries”. The operation prefix (for example, select;, insert;, update;, delete; - see the native procedure logic below) must be present in the native query, but it will not be issued as part of the query to the source.

Example 13.11. Example DDL for a SF native procedure

```
CREATE FOREIGN PROCEDURE proc (arg1 integer, arg2 string) OPTIONS
("teiid_rel:native-query" 'search;SELECT ... complex SOQL ... WHERE col1
= $1 and col2 = $2') returns (col1 string, col2 string, col3 timestamp);
```

13.25.12. Salesforce Translator: Native Procedure

WARNING

This feature is turned off by default because of the security risk this exposes to execute any command against the source. To enable this feature, override the translator property called "SupportsNativeQueries" to true. See Section 13.6, “Override Execution Properties”.

SalesForce translator provides a procedure with name native that gives ability to execute any ad hoc native Salesforce queries directly against the source without any JBoss Data Virtualization parsing or resolving. The metadata of this procedure's execution results are not known to JBoss Data Virtualization, and they are returned as object array. User can use an ARRAYTABLE construct (Section 3.6.9, “Nested Tables: ARRAYTABLE”) to build a tabular output for consumption by client applications. JBoss Data Virtualization exposes this procedure with a simple query structure as below.

13.25.13. Salesforce Translator Example: Select

Example 13.12. Select Example

```
SELECT x.* FROM (call pm1.native('search;SELECT Account.Id,
Account.Type, Account.Name FROM Account')) w,
 ARRAYTABLE(w.tuple COLUMNS "id" string , "type" string, "name" String)
AS x
```

In the above code, the "search" keyword is followed by a query statement.

NOTE

The Salesforce Object Query Language (SOQL) is treated as a parameterized native query so that parameter values may be inserted in the query string properly. See Section 13.7, "Parameterizable Native Queries".

The results returned by search may contain the object Id as the first column value regardless of whether it was selected. Also queries that select columns from multiple object types will not be correct.

13.25.14. Salesforce Translator Example: Delete

Example 13.13. Delete Example

```
SELECT x.* FROM (call pm1.native('delete;', 'id1', 'id2')) w,
 ARRAYTABLE(w.tuple COLUMNS "updatecount" integer) AS x
```

In the above code, the "delete;" keyword is followed by the ids to delete as varargs.

13.25.15. Salesforce Translator Example: Create and Update

Example 13.14. Create Example

```
SELECT x.* FROM
 (call pm1.native('create;type=table;attributes=one,two,three', 'one',
2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

In the above code, the "create" or "update" keyword must be followed by the following properties. Attributes must be matched positionally by the procedure variables - thus in the example attribute two will be set to 2.

Property Name	Description	Required
type	Table Name	Yes
attributes	comma separated list of names of the columns	

The values for each attribute is specified as separate argument to the "native" procedure.

Update is similar to create, with one more extra property called "id", which defines identifier for the record.

Example 13.15. Update Example

```
SELECT x.* FROM
 (call pm1.native('update;id=pk;type=table;attributes=one,two,three',
'one', 2, 3.0)) w,
 ARRAYTABLE(w.tuple COLUMNS "update_count" integer) AS x
```

IMPORTANT

By default the name of the procedure that executes the queries directly is called native, however user can set override execution property **vdb.xml** file to change it.

13.26. SAP GATEWAY TRANSLATOR

Teiid provides a translator for SAP Gateway using the OData protocol. This translator is extension of OData Translator and uses Teiid WS resource adapter for making web service calls. This translator understands the most of the SAP specific OData extensions to the metadata.

When the metadata is imported from SAP Gateway, the Teiid models are created to accordingly for SAP specific EntitySet and Property annotations.

These "execution properties" are supported in this translator:

Table 13.23. Execution Properties

Property	Description	Default
DatabaseTimeZone	The time zone of the database. Used when fetchings date, time, or timestamp values	The system default time zone
SupportsOdataCount	Supports $count	True
SupportsOdataFilter	Supports $filter	True
SupportsOdataOrderBy	Supports $orderby	True
SupportsOdataSkip	Supports $skip	True
SupportsOdataTop	Supports $top	True

WARNING

If metadata on your service defined "pagable" and/or "topable" as "false' on any table, you must turn off "SupportsOdataTop" and "SupportsOdataSkip" execution-properties in your translator, so that you will not end up with wrong results. SAP metadata has capability to control these in a fine grained fashion any on EnitySet, however Teiid can only control these at translator level.

WARNING

Sample examples defined at http://scn.sap.com/docs/DOC-31221, we found to be lacking in full metadata in certain examples. For example, "filterable" clause never defined on some properties, but if you send a request $filter it will silently ignore it. You can verify this behavior by directly executing the REST service using a web browser with respective query. So, Make sure you have implemented your service correctly, or you can turn off certain features in this translator by using "execution properties" override.

13.27. WEB SERVICES TRANSLATOR

13.27.1. Web Services Translator

The Web Services translator exposes stored procedures for calling web services.

The Web Services translator is implemented by the `org.teiid.translator.ws.WSExecutionFactory` class and known by the translator type name `ws`.

The corresponding resource adapter may optionally be configured to point at a specific WSDL. Results from this translator will commonly be used with the TEXTTABLE or XMLTABLE table functions to use CSV or XML formatted data. See Section 3.6.7, “Nested Tables: TEXTTABLE” and Section 3.6.8, “Nested Tables: XMLTABLE” for more details.

There are no importer settings for this translator, but it does provide metadata for dynamic VDBs. If the connection is configured to point at a specific WSDL, the translator will import all SOAP operations under the specified service and port as procedures.

NOTE

The resource adapter for this translator is provided by configuring the **webservices** data source in the JBoss EAP instance. See the *Red Hat JBoss Data Virtualization Administration and Configuration Guide* for more configuration information.

IMPORTANT

Setting the proper binding value on the translator is recommended as it removes the need for callers to pass an explicit value. If your service actually uses SOAP11, but the binding used SOAP12 you will receive execution failures.

13.27.2. Web Services Translator: Execution Properties

Table 13.24. Execution Properties

Name	Description	When Used	Default
DefaultBinding	The binding that should be used if one is not specified. Can be one of HTTP, SOAP11, or SOAP12	invoke*	SOAP12
DefaultServiceMode	The default service mode. For SOAP, MESSAGE mode indicates that the request will contain the entire SOAP envelope and not just the contents of the SOAP body. Can be one of MESSAGE or PAYLOAD.	invoke* or WSDL call	PAYLOAD
XMLParamName	Used with the HTTP binding (typically with the GET method) to indicate that the request document should be part of the query string.	invoke*	null - unused

IMPORTANT

If you want to expose a virtual stored procedure as a SOAP web service which must implement Basic Auth, you will encounter an exception unless you set this property:

```
<validate-on-match>true</validate-on-match>
```

13.27.3. Web Services Translator: Usage

The WS translator exposes two low level procedures for accessing web services: invoke and invokeHttp.

13.27.4. Web Services Translator: Invoke Procedure

Invoke allows for multiple binding, or protocol modes, including HTTP, SOAP11, and SOAP12.

```
Procedure invoke(binding in STRING, action in STRING, request in XML,
endpoint in STRING) returns XML
```

The binding may be one of null (to use the default) HTTP, SOAP11, or SOAP12. Action with a SOAP binding indicates the SOAPAction value. Action with a HTTP binding indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for the binding or endpoint will use the default value. The default endpoint is specified in the WS resource adapter configuration. The endpoint URL may be absolute or relative. If it is relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invoke procedure with named parameter syntax.

```
call invoke(binding=>'HTTP', action=>'GET')
```

The request XML should be a valid XML document or root element.

If the stream parameter is set to true, the resulting value document can only be read once. This is appropriate when directly passing the XML into XMLQUERY or XMLTABLE and only a single pass against the document is needed. If stream is null or false, then the engine may need to save a copy of the document for repeated use.

13.27.5. Web Services Translator: InvokeHTTP Procedure

`invokeHttp` can return the byte contents of an HTTP or HTTPS call.

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in
STRING, contentType out STRING) returns BLOB
```

Action indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for endpoint will use the default value. The default endpoint is specified in the WS resource adapter configuration. The endpoint URL may be absolute or relative. If it is relative then it will be combined with the default endpoint.

Since multiple parameters are not required to have values, it is often more clear to call the invoke procedure with named parameter syntax.

```
call invokeHttp(action=>'GET')
```

The request can be one of SQLXML, STRING, BLOB, or CLOB. The request will be sent as the POST payload in byte form. For STRING/CLOB values this will default to the UTF-8 encoding. Use the **TO_BYTES** function to control the byte encoding.

If the stream parameter is set to true, then the resulting lob value may only be used a single time. If stream is null or false, then the engine may need to save a copy of the result for repeated use. Care must be used as some operations, such as casting or XMLPARSE may perform validation which results in the stream being consumed.

The resource adapter for this translator is a Web Service Data Source.

IMPORTANT

Currently you can only use WSDL based Procedures if they participate in WS-Security, when resource-adapter is configured with correct CXF configuration.

CHAPTER 14. FEDERATED PLANNING

14.1. FEDERATED PLANNING

At the core of JBoss Data Virtualization is a federated relational query engine. This query engine allows you to treat all of your data sources as one virtual database and access them in a single SQL query. This allows you to focus on building your application, rather than on manually coding joins, and other relational operations, between data sources.

14.2. PLANNING OVERVIEW

When the query engine receives an incoming SQL query it performs the following operations.

1. Parsing - syntax is validated and converted to internal form.
2. Resolving - all identifiers are linked to metadata, and functions are linked to the function library.
3. Validating - SQL semantics are validated based on metadata references and type signatures.
4. Rewriting - SQL is rewritten to simplify expressions and criteria.
5. Logical plan optimization - the rewritten canonical SQL is converted into a logical plan for in-depth optimization. The JBoss Data Virtualization optimizer is predominantly rule-based. Based upon the query structure and hints, a certain rule set will be applied. These rules may in turn trigger the execution of more rules. Within several rules, JBoss Data Virtualization also takes advantage of costing information. The logical plan optimization steps can be seen by using the SHOWPLAN DEBUG clause and are described in Section 14.9.1, “Query Planner”.
6. Processing plan conversion - the logic plan is converted into an executable form where the nodes are representative of basic processing operations. The final processing plan is displayed as the query plan. See Section 14.8.1, “Query Plans”.

The logical query plan is a tree of operations used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output).

The primary logical operations and their SQL equivalents are:

- *select* - select or filter rows based on a criteria,
- *project* - project or compute column values,
- *join*,
- *source* - retrieve data from a table,
- *sort* - ORDER BY,
- *duplicate removal* - SELECT DISTINCT,
- *group* - GROUP BY, and
- *union* - UNION.

14.3. EXAMPLE QUERY

The following example has a query that retrieves all engineering employees born since 1970.

Example 14.1. Example query

```
SELECT e.title, e.lastname FROM Employees AS e JOIN
Departments AS d ON e.dept_id = d.dept_id WHERE year(e.birthday) >= 1970
AND d.dept_name = 'Engineering'
```

Logically, the data from the Employees and Departments tables are retrieved, then joined, then filtered as specified, and finally the output columns are projected. The canonical query plan thus looks like this:

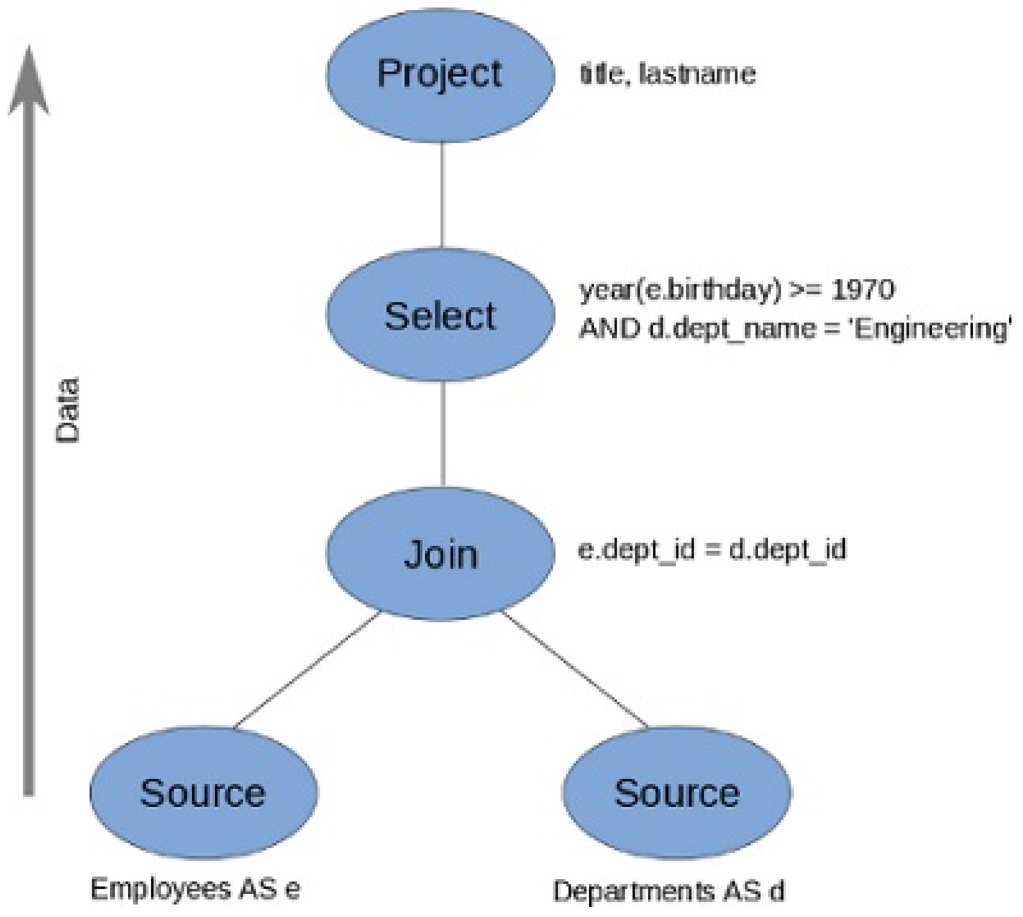

Figure 14.1. Canonical Query Plan

Data flows from the tables at the bottom upwards through the join, through the select, and finally through the project to produce the final results. The data passed between each node is logically a result set with columns and rows.

This is what happens *logically* , not how the plan is actually executed. Starting from this initial plan, the query planner performs transformations on the query plan tree to produce an equivalent plan that retrieves the same results faster. Both a federated query planner and a relational database planner deal with the same concepts and many of the same plan transformations. In this example, the criteria on the Departments and Employees tables will be pushed down the tree to filter the results as early as possible.

In both cases, the goal is to retrieve the query results in the fastest possible time. However, the relational database planner does this primarily by optimizing the access paths in pulling data from storage.

In contrast, a federated query planner is less concerned about storage access because it is typically pushing that burden to the data source. The most important consideration for a federated query planner is minimizing data transfer.

14.4. SUBQUERY OPTIMIZATION

- EXISTS subqueries are typically rewrite to "SELECT 1 FROM ..." to prevent unnecessary evaluation of SELECT expressions.

- Quantified compare SOME subqueries are always turned into an equivalent IN predicate or comparison against an aggregate value. e.g. col > SOME (select col1 from table) would become col > (select min(col1) from table)

- Uncorrelated EXISTs and scalar subquery that are not pushed to the source can be evaluated prior to source command formation.

- Correlated subqueries used in DELETEs or UPDATEs that are not pushed as part of the corresponding DELETE/UPDATE will cause JBoss Data Virtualization to perform row-by-row compensating processing. This will only happen if the affected table has a primary key. If it does not, then an exception will be thrown.

- WHERE or HAVING clause IN, Quantified Comparison, Scalar Subquery Compare, and EXISTs predicates can take the MJ (merge join), DJ (dependent join), or NO_UNNEST (no unnest) hints appearing just before the subquery. The MJ hint directs the optimizer to use a traditional, semijoin, or antisemijoin merge join if possible. The DJ is the same as the MJ hint, but additionally directs the optimizer to use the subquery as the independent side of a dependent join if possible. The NO_UNNEST hint, which supercedes the other hints, will direct the optimizer to leave the subquery in place.

  Example 14.2. Merge Join Hint Usage

  ```
  SELECT col1 from tbl where col2 IN /*+ MJ */ (SELECT col1 FROM
  tbl2)
  ```

  Example 14.3. Dependent Join Hint Usage

  ```
  SELECT col1 from tbl where col2 IN /*+ DJ */ (SELECT col1 FROM
  tbl2)
  ```

  Example 14.4. No Unnest Hint Usage

  ```
  SELECT col1 from tbl where col2 IN /*+ NO_UNNEST */ (SELECT col1
  FROM tbl2)
  ```

- The system property org.teiid.subqueryUnnestDefault controls whether the optimizer will by default unnest subqueries. If true, then most non-negated WHERE or HAVING clause non-negated EXISTS or IN subquery predicates can be converted to a traditional join.

- The planner will always convert to anitjoin or semijoin vartiants is costing is favorable. Use a hint to override this behavior if needed.

- EXISTs and scalar subqueries that are not pushed down, and not converted to merge joins, are implicitly limited to 1 and 2 result rows respectively.
- Conversion of subquery predicates to nested loop joins is not yet available.

14.5. XQUERY OPTIMIZATION

A technique known as document projection is used to reduce the memory footprint of the context item document. Document projection loads only the parts of the document needed by the relevant XQuery and path expressions. Since document projection analysis uses all relevant path expressions, even 1 expression that could potentially use many nodes, e.g. //x rather than /a/b/x will cause a larger memory footprint. With the relevant content removed the entire document will still be loaded into memory for processing. Document projection will only be used when there is a context item (unnamed PASSING clause item) passed to XMLTABLE/XMLQUERY. A named variable will not have document projection performed. In some cases the expressions used may be too complex for the optimizer to use document projection. You should check the SHOWPLAN DEBUG full plan output to see if the appropriate optimization has been performed.

With additional restrictions, simple context path expressions allow the processor to evaluate document subtrees independently - without loading the full document in memory. A simple context path expression can be of the form "[/][ns:]root/[ns1:]elem/...", where a namespace prefix or element name can also be the * wild card. As with normal XQuery processing, if namespace prefixes are used in the XQuery expression, they should be declared using the XMLNAMESPACES clause.

Example 14.5. Streaming Eligible XMLQUERY

```
XMLQUERY('/*:root/*:child' PASSING doc)
```

Rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Example 14.6. Streaming Ineligible XMLQUERY

```
XMLQUERY('//child' PASSING doc)
```

The use of the descendant axis prevents the streaming optimization, but document projection can still be performed.

When using XMLTABLE, the COLUMN PATH expressions have additional restrictions. They are allowed to reference any part of the element subtree formed by the context expression and they may use any attribute value from their direct parentage. Any path expression where it is possible to reference a non-direct ancestor or sibling of the current context item prevent streaming from being used.

Example 14.7. Streaming Eligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS fullchild XML PATH '.',
parent_attr string PATH '../@attr', child_val integer)
```

The context XQuery and the column path expression allow the streaming optimization, rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Example 14.8. Streaming Ineligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS sibling_attr string PATH
'../other_child/@attr')
```

The reference of an element outside of the child subtree in the sibling_attr path prevents the streaming optimization from being used, but document projection can still be performed.

Column paths should be as targeted as possible to avoid performance issues. A general path such as '..//child' will cause the entire subtree of the context item to be searched on each output row.

14.6. PARTIAL RESULTS

JBoss Data Virtualization provides the capability to obtain "partial results" in the event of data source unavailability or failure. This is especially useful when unioning information from multiple sources, or when doing a left outer join, where you are 'appending' columns to a master record but still want the record if the extra information is not available.

A source is considered to be 'unavailable' if the connection factory associated with the source issues an exception in response to a query. The exception will be propagated to the query processor, where it will become a warning on the statement. See *Red Hat JBoss Data Virtualization Development Guide: Client Development* for more on Partial Results Mode and SQLWarnings.

14.7. FEDERATED OPTIMIZATIONS

14.7.1. Access Patterns

Access patterns are used on both physical tables and views to specify the need for criteria against a set of columns. Failure to supply the criteria will result in a planning error, rather than a runaway source query. Access patterns can be applied in a set such that only one of the access patterns is required to be satisfied.

Currently any form of criteria referencing an affected column may satisfy an access pattern.

14.7.2. Pushdown

In federated database systems, pushdown refers to decomposing the user query into source queries that perform as much work as possible on their respective source system. Pushdown analysis requires knowledge of source system capabilities, which is provided to JBoss Data Virtualization though the Connector API. Any work not performed at the source is then processed in the federating system's relational engine (in JBoss Data Virtualization).

Based upon capabilities, JBoss Data Virtualization will manipulate the query plan to ensure that each source performs as much joining, filtering, grouping, etc. as possible. In many cases, such as with join ordering, planning combines standard relational techniques (see Section 14.7.9, “Standard Relational Techniques”) and heuristics based on cost effectiveness to optimize pushdowns.

Criteria and join push down are typically the most important aspects of the query to push down when performance is a concern. See Section 14.8.1, “Query Plans” for information about how to read a plan to ensure that source queries are as efficient as possible.

14.7.3. Dependent Joins

A special optimization called a dependent join is used to reduce the rows returned from one of the two relations involved in a multi-source join. In a dependent join, queries are issued to each source sequentially rather than in parallel, with the results obtained from the first source used to restrict the records returned from the second. Dependent joins can perform some joins much faster by reducing the amount of data retrieved from the second source and the number of join comparisons that must be performed.

The conditions when a dependent join is used are determined by the query planner based on access patterns, hints, and costing information. There are three different kinds of dependent joins that Teiid supports:

- Join based on in/equality support: where the engine will determine how to break of the queries
- Key Pushdown: where the translator has access to the full set of key values and determines what queries to send
- Full Pushdown - where translator ships the all data from the independent side to the translator. Can be used automatically by costing or can be specified as an option in the hint.

JBoss Data Virtualization supports hints to control dependent join behavior:

- MAKEIND - indicates that the clause should be the independent side of a dependent join.
- MAKEDEP - indicates that the clause should be the dependent side of a join. MAKEDEP as a non-comment hint supports optional max and join arguments - MAKEDEP(JOIN) meaning that the entire join should be pushed, and MAKEDEP(MAX:5000) meaning that the dependent join should only be performed if there are less than the max number of values from the independent side.
- MAKENOTDEP - prevents the clause from being the dependent side of a join.

These can be placed in either the OPTION clause or directly in the FROM clause. As long as all access patterns can be met, the MAKEIND, MAKEDEP, and MAKENOTDEP hints override any use of costing information. MAKENOTDEP supersedes the other hints.

NOTE

The MAKEDEP/MAKEIND hint must only be used if the proper query plan is not chosen by default. Ensure that your costing information is representative of the actual source cardinality. An inappropriate MAKEDEP/MAKEIND hint can force an inefficient join structure and may result in many source queries.

For IN clauses, the engine will filter the values coming from the dependent side. If the number of values from the independent side exceeds the translators MaxInCriteriaSize, the values will be split into multiple IN predicates up to MaxDependentPredicates. When the number of independent values exceeds MaxInCriteriaSize*MaxDependentPredicates, then multiple dependent queries will be issued in parallel.

NOTE

While these hints can be applied to views, the optimizer will by default remove views when possible. This can result in the hint placement being significantly different than that which was originally intended. Consider using the NO_UNNEST hint to prevent the optimizer from removing the view in these cases.

A "full pushdown", sometimes also called a "data-ship pushdown", is where all the data from independent side of the join is sent to dependent side. Currently this is only supported in the JDBC translators. To enable it, provide translator override property "enableDependentJoins" to "true". The JDBC source must support creation temp tables (this is determined by using Hibernate dialect capabilities for the source). Once these properties are enabled and MAKEDEP hint is used, the translator will ship the data as temp table contents and push the dependent join to the source for full processing.

14.7.4. Copy Criteria

Copy criteria is an optimization that creates additional predicates based upon combining join and where clause criteria. For example, equi-join predicates (source1.table.column = source2.table.column) are used to create new predicates by substituting source1.table.column for source2.table.column and vice versa. In a cross source scenario, this allows for WHERE criteria applied to a single side of the join to be applied to both source queries.

14.7.5. Projection Minimization

JBoss Data Virtualization ensures that each pushdown query only projects the symbols required for processing the user query. This is especially helpful when querying through large intermediate view layers.

14.7.6. Partial Aggregate Pushdown

Partial aggregate pushdown allows for grouping operations above multi-source joins and unions to be decomposed so that some of the grouping and aggregate functions may be pushed down to the sources.

14.7.7. Optional Join

The optional join hint indicates to omit a joined table if none of its columns are used by the output of the user query or in a meaningful way to construct the results of the user query. This hint is typically only used in view layers containing multi-source joins.

The optional join hint is applied as a comment on a join clause. It can be applied in both ANSI and non-ANSI joins. With non-ANSI joins an entire joined table may be marked as optional.

Example 14.9. Example Optional Join Hint

```
select a.column1, b.column2 from a, /*+ optional */ b WHERE a.key =
b.key
```

Suppose this example defines a view layer X. If X is queried in such a way as to not need b.column2, then the optional join hint will cause b to be omitted from the query plan. The result would be the same as if X were defined as:

```
select a.column1 from a
```

Example 14.10. Example ANSI Optional Join Hint

```
select a.column1, b.column2, c.column3 from /*+ optional */ (a inner
join b ON a.key = b.key) INNER JOIN c ON a.key = c.key
```

In this example the ANSI join syntax allows for the join of a and b to be marked as optional. Suppose this example defines a view layer X. Only if both column a.column1 and b.column2 are not needed, e.g. "SELECT column3 FROM X" will the join be removed.

The optional join hint will not remove a bridging table that is still required.

Example 14.11. Example Bridging Table

```
select a.column1, b.column2, c.column3 from /*+ optional */ a, b, c
WHERE ON a.key = b.key AND a.key = c.key
```

Suppose this example defines a view layer X. If b.column2 or c.column3 are solely required by a query to X, then the join on a can be removed. However if a.column1 or both b.column2 and c.column3 are needed, then the optional join hint will not take effect.

NOTE

When a join clause is omitted via the optional join hint, the relevant criteria is not applied. Thus it is possible that the query results may not have the same cardinality or even the same row values as when the join is fully applied.

Left/right outer joins where the inner side values are not used and whose rows under go a distinct operation will automatically be treated as an optional join and do not require a hint.

Example 14.12. Example Unnecessary Optional Join Hint

```
select a.column1, b.column2 from a LEFT OUTER JOIN
/*+optional*/ b ON a.key = b.key
```

WARNING

A simple "SELECT COUNT(*) FROM VIEW" against a view where all join tables are marked as optional will not return a meaningful result.

Source Hints

Teiid user and transformation queries can contain a meta source hint that can provide additional information to source queries. The source hint has the form:

```
/*+ sh[[ KEEP ALIASES]:'arg'] source-name[ KEEP ALIASES]:'arg1' ... */
```

The source hint is expected to appear after the query (SELECT, INSERT, UPDATE, DELETE) keyword.

Source hints may appear in any subquery or in views. All hints applicable to a given source query will be collected and pushed down together as a list. The order of the hints is not guaranteed.

The sh arg is optional and is passed to all source queries via the ExecutionContext.getGeneralHints method. The additional args should have a source-name that matches the source name assigned to the translator in the VDB.

If the source-name matches, the hint values will be supplied via the ExecutionContext.getSourceHints method.

Each of the arg values has the form of a string literal - it must be surrounded in single quotes and a single quote can be escaped with another single quote. Only the Oracle translator does anything with source hints by default. The Oracle translator will use both the source hint and the general hint (in that order) if available to form an Oracle hint enclosed in /*+ ... */.

If the KEEP ALIASES option is used either for the general hint or on the applicable source specific hint, then the table/view aliases from the user query and any nested views will be preserved in the push-down query. This is useful in situations where the source hint may need to reference aliases and the user does not wish to rely on the generated aliases (which can be seen in the query plan in the relevant source queries - see above). However in some situations this may result in an invalid source query if the preserved alias names are not valid for the source or result in a name collision. If the usage of KEEP ALIASES results in an error, the query could be modified by preventing view removal with the NO_UNNEST hint, the aliases modified, or the KEEP ALIASES option could be removed and the query plan used to determine the generated alias names.

Here are some sample source hints:

```
SELECT /*+ sh:'general hint' */ ...
```

```
SELECT /*+ sh KEEP ALIASES:'general hint' my-oracle:'oracle hint' */ ...
```

14.7.8. Partitioned Union

Union partitioning is inferred from the transformation/inline view. If one (or more) of the UNION columns is defined by constants and/or has WHERE clause IN predicates containing only constants that make each branch mutually exclusive, then the UNION is considered partitioned. UNION ALL must be used and the UNION cannot have a LIMIT, WITH, or ORDER BY clause (although individual branches may use LIMIT, WITH, or ORDER BY). Partitioning values should not be null.

For example the view definition "select 1 as x, y from foo union all select z, a from foo1 where z in (2, 3)" would be considered partitioned on column x, since the first branch can only be the value 1 and the second branch can only be the values 2 or 3.

NOTE

More advanced or explicit partitioning could be considered in the future. The concept of a partitioned union is used for performing partition-wise joins (see Section 5.1, "Updatable Views" and Section 14.7.6, "Partial Aggregate Pushdown").

14.7.9. Standard Relational Techniques

JBoss Data Virtualization also incorporates many standard relational techniques to ensure efficient query plans.

- Rewrite analysis for function simplification and evaluation.
- Boolean optimizations for basic criteria simplification.
- Removal of unnecessary view layers.
- Removal of unnecessary sort operations.
- Advanced search techniques through the left-linear space of join trees.
- Parallelizing of source access during execution.
- Subquery optimization (Section 14.4, “Subquery Optimization”)

14.8. QUERY PLANS

14.8.1. Query Plans

When integrating information using a federated query planner, it is useful to be able to view the query plans that are created, to better understand how information is being accessed and processed, and to troubleshoot problems.

A query plan is a set of instructions created by a query engine for executing a command submitted by a user or application. The purpose of the query plan is to execute the user's query in as efficient a way as possible.

14.8.2. Getting a Query Plan

You can get a query plan any time you execute a command. The SQL options available are as follows:

- SET SHOWPLAN [ON|DEBUG]- Returns the processing plan or the plan and the full planner debug log.

With the above options, the query plan is available from the Statement object by casting to the **`org.teiid.jdbc.TeiidStatement`** interface or by using the "SHOW PLAN" statement.

Example 14.13. Retrieving a Query Plan

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
TeiidStatement tstatement = statement.unwrap(TeiidStatement.class);
PlanNode queryPlan = tstatement.getPlanDescription();
System.out.println(queryPlan);
```

The query plan is made available automatically in several JBoss Data Virtualization tools.

14.8.3. Analyzing a Query Plan

Once a query plan has been obtained you will most commonly be looking for:

- Source pushdown - what parts of the query were pushed to each source? Ensure that any predicates, especially against, indexes are pushed.

- Join ordering - as federated joins can be quite expensive. They are typically influenced by costing.
- Join criteria type mismatches.
- Join algorithm used - merge, enhanced merge, nested loop and so forth.
- Presence of federated optimizations, such as dependent joins.
- Join criteria type mismatches.

All of these issues presented above will be present subsections of the plan that are specific to relational queries. If you are executing a procedure or generating an XML document, the overall query plan will contain additional information related the surrounding procedural execution.

A query plan consists of a set of nodes organized in a tree structure. As with the above example, you will typically be interested in analyzing the textual form of the plan.

In a procedural context the ordering of child nodes implies the order of execution. In most other situation, child nodes may be executed in any order even in parallel. Only in specific optimizations, such as dependent join, will the children of a join execute serially.

14.8.4. Relational Plans

Relational plans represent the actually processing plan that is composed of nodes that are the basic building blocks of logical relational operations. Physical relational plans differ from logical relational plans in that they will contain additional operations and execution specifics that were chosen by the optimizer.

The nodes for a relational query plan are:

Access

Access a source. A source query is sent to the connection factory associated with the source. [For a dependent join, this node is called Dependent Access.]

Dependent Procedure Access

Access a stored procedure on a source using multiple sets of input values.

Batched Update

Processes a set of updates as a batch.

Project

Defines the columns returned from the node. This does not alter the number of records returned.

Project Into

Like a normal project, but outputs rows into a target table.

Select

Select is a criteria evaluation filter node (WHERE / HAVING). When there is a subquery in the criteria, this node is called Dependent Select.

Insert Plan Execution

Similar to a project into, but executes a plan rather than a source query. Typically created when executing an insert into view with a query expression.

Window Function Project

Like a normal project, but includes window functions.

Select

Select is a criteria evaluation filter node (WHERE/HAVING).

Join

Defines the join type, join criteria, and join strategy (merge or nested loop).

Union All

There are no properties for this node; it just passes rows through from its children. Depending upon other factors, such as if there is a transaction or the source query concurrency allowed, not all of the union children will execute in parallel.

Sort

Defines the columns to sort on, the sort direction for each column, and whether to remove duplicates or not.

Dup Remove

Removes duplicate rows. The processing uses a tree structure to detect duplicates so that results will effectively stream at the cost of IO operations.

Grouping

Groups sets of rows into groups and evaluates aggregate functions.

Null

A node that produces no rows. Usually replaces a Select node where the criteria is always false (and whatever tree is underneath). There are no properties for this node.

Plan Execution

Executes another sub plan. Typically the sub plan will be a non-relational plan.

Dependent Procedure Execution

Executes a sub plan using multiple sets of input values.

Limit

Returns a specified number of rows, then stops processing. Also processes an offset if present.

XML Table

Evaluates XMLTABLE. The debug plan will contain more information about the XQuery/XPath with regards to their optimization - see the XQuery section below or XQuery Optimization.

Text Table

Evaluates TEXTTABLE

Array Table

Evaluates ARRAYTABLE

Object Table

Evaluates OBJECTTABLE

14.8.5. Relational Plans: Node Statistics

Every node has a set of statistics that are output. These can be used to determine the amount of data flowing through the node. Before execution a processor plan will not contain node statistics. Also the statistics are updated as the plan is processed, so typically you will want the final statistics after all rows have been processed by the client.

Table 14.1. Node Statistics

Statistic	Description	Units
Node Output Rows	Number of records output from the node	count
Node Next Batch Process Time	Time processing in this node only	millisec
Node Cumulative Process Time	Elapsed time from beginning of processing to end	millisec
Node Cumulative Next Batch Process Time	Time processing in this node + child nodes	millisec
Node Next Batch Calls	Number of times a node was called for processing	count
Node Blocks	Number of times a blocked exception was thrown by this node or a child	count

In addition to node statistics, some nodes display cost estimates computed at the node.

Table 14.2. Node Cost Estimates

Cost Estimates	Description	Units
Estimated Node Cardinality	Estimated number of records that will be output from the node; -1 if unknown	count

14.8.6. Source Hints

The root node will display additional information.

Table 14.3. Registry Properties

Top level Statistics	Description	Units
Data Bytes Sent	The size of the serialized data result (row and lob values) sent to the client	bytes

The query processor plan can be obtained in a plain text or xml format. The plan text format is typically easier to read, while the xml format is easier to process by tooling. When possible tooling should be used to examine the plans as the tree structures can be deeply nested.

Data flows from the leafs of the tree to the root. Sub plans for procedure execution can be shown inline, and are differentiated by different indentation. Given a user query of "SELECT pm1.g1.e1, pm1.g2.e2, pm1.g3.e3 from pm1.g1 inner join (pm1.g2 left outer join pm1.g3 on pm1.g2.e1=pm1.g3.e1) on pm1.g1.e1=pm1.g3.e1" the text for a processor plan that does not push down the joins would look like:

```
ProjectNode
  + Output Columns:
    0: e1 (string)
    1: e2 (integer)
    2: e3 (boolean)
  + Cost Estimates:Estimated Node Cardinality: -1.0
  + Child 0:
    JoinNode
      + Output Columns:
        0: e1 (string)
        1: e2 (integer)
        2: e3 (boolean)
      + Cost Estimates:Estimated Node Cardinality: -1.0
      + Child 0:
        JoinNode
          + Output Columns:
            0: e1 (string)
            1: e1 (string)
            2: e3 (boolean)
          + Cost Estimates:Estimated Node Cardinality: -1.0
          + Child 0:
            AccessNode
              + Output Columns:e1 (string)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0 FROM pm1.g1 AS g_0 ORDER BY c_0
              + Model Name:pm1
          + Child 1:
            AccessNode
              + Output Columns:
                0: e1 (string)
                1: e3 (boolean)
              + Cost Estimates:Estimated Node Cardinality: -1.0
              + Query:SELECT g_0.e1 AS c_0, g_0.e3 AS c_1 FROM pm1.g3 AS
g_0 ORDER BY c_0
              + Model Name:pm1
          + Join Strategy:MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
          + Join Type:INNER JOIN
          + Join Criteria:pm1.g1.e1=pm1.g3.e1
      + Child 1:
        AccessNode
```

```
            + Output Columns:
              0: e1 (string)
              1: e2 (integer)
            + Cost Estimates:Estimated Node Cardinality: -1.0
            + Query:SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM pm1.g2 AS g_0
ORDER BY c_0
            + Model Name:pm1
        + Join Strategy:ENHANCED SORT JOIN (SORT/ALREADY_SORTED)
        + Join Type:INNER JOIN
        + Join Criteria:pm1.g3.e1=pm1.g2.e1
    + Select Columns:
      0: pm1.g1.e1
      1: pm1.g2.e2
      2: pm1.g3.e3
```

Note that the nested join node is using a merge join and expects the source queries from each side to produce the expected ordering for the join. The parent join is an enhanced sort join which can delay the decision to perform sorting based upon the incoming rows. Note that the outer join from the user query has been modified to an inner join since none of the null inner values can be present in the query result.

The same plan in xml form looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="ProjectNode">
    <property name="Output Columns">
        <value>e1 (string)</value>
        <value>e2 (integer)</value>
        <value>e3 (boolean)</value>
    </property>
    <property name="Cost Estimates">
        <value>Estimated Node Cardinality: -1.0</value>
    </property>
    <property name="Child 0">
        <node name="JoinNode">
            <property name="Output Columns">
                <value>e1 (string)</value>
                <value>e2 (integer)</value>
                <value>e3 (boolean)</value>
            </property>
            <property name="Cost Estimates">
                <value>Estimated Node Cardinality: -1.0</value>
            </property>
            <property name="Child 0">
                <node name="JoinNode">
                    <property name="Output Columns">
                        <value>e1 (string)</value>
                        <value>e1 (string)</value>
                        <value>e3 (boolean)</value>
                    </property>
                    <property name="Cost Estimates">
                        <value>Estimated Node Cardinality: -1.0</value>
                    </property>
                    <property name="Child 0">
                        <node name="AccessNode">
                            <property name="Output Columns">
```

```
                                <value>e1 (string)</value>
                            </property>
                            <property name="Cost Estimates">
                                <value>Estimated Node Cardinality: -
1.0</value>
                            </property>
                            <property name="Query">
                                <value>SELECT g_0.e1 AS c_0 FROM pm1.g1
AS g_0 ORDER BY c_0</value>
                            </property>
                            <property name="Model Name">
                                <value>pm1</value>
                            </property>
                        </node>
                    </property>
                    <property name="Child 1">
                        <node name="AccessNode">
                            <property name="Output Columns">
                                <value>e1 (string)</value>
                                <value>e3 (boolean)</value>
                            </property>
                            <property name="Cost Estimates">
                                <value>Estimated Node Cardinality: -
1.0</value>
                            </property>
                            <property name="Query">
                                <value>SELECT g_0.e1 AS c_0, g_0.e3 AS
c_1 FROM pm1.g3 AS g_0
                                    ORDER BY c_0</value>
                            </property>
                            <property name="Model Name">
                                <value>pm1</value>
                            </property>
                        </node>
                    </property>
                    <property name="Join Strategy">
                        <value>MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
</value>
                    </property>
                    <property name="Join Type">
                        <value>INNER JOIN</value>
                    </property>
                    <property name="Join Criteria">
                        <value>pm1.g1.e1=pm1.g3.e1</value>
                    </property>
                </node>
            </property>
            <property name="Child 1">
                <node name="AccessNode">
                    <property name="Output Columns">
                        <value>e1 (string)</value>
                        <value>e2 (integer)</value>
                    </property>
                    <property name="Cost Estimates">
                        <value>Estimated Node Cardinality: -1.0</value>
                    </property>
                    <property name="Query">
                        <value>SELECT g_0.e1 AS c_0, g_0.e2 AS c_1 FROM
pm1.g2 AS g_0
                            ORDER BY c_0</value>
                    </property>
                    <property name="Model Name">
                        <value>pm1</value>
                    </property>
                </node>
            </property>
            <property name="Join Strategy">
                <value>ENHANCED SORT JOIN (SORT/ALREADY_SORTED)</value>
            </property>
            <property name="Join Type">
                <value>INNER JOIN</value>
            </property>
            <property name="Join Criteria">
                <value>pm1.g3.e1=pm1.g2.e1</value>
            </property>
        </node>
    </property>
    <property name="Select Columns">
        <value>pm1.g1.e1</value>
        <value>pm1.g2.e2</value>
        <value>pm1.g3.e3</value>
    </property>
</node>
```

Note that the same information appears in each of the plan forms. In some cases it can actually be easier to follow the simplified format of the debug plan final processor plan. From the Debug Log the same plan as above would appear as:

```
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
ProjectNode(0) output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3] [pm1.g1.e1,
pm1.g2.e2, pm1.g3.e3]
  JoinNode(1) [ENHANCED SORT JOIN (SORT/ALREADY_SORTED)] [INNER JOIN]
criteria=[pm1.g3.e1=pm1.g2.e1] output=[pm1.g1.e1, pm1.g2.e2, pm1.g3.e3]
    JoinNode(2) [MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)] [INNER JOIN]
criteria=[pm1.g1.e1=pm1.g3.e1] output=[pm1.g3.e1, pm1.g1.e1, pm1.g3.e3]
      AccessNode(3) output=[pm1.g1.e1] SELECT g_0.e1 AS c_0 FROM pm1.g1 AS
g_0 ORDER BY c_0
      AccessNode(4) output=[pm1.g3.e1, pm1.g3.e3] SELECT g_0.e1 AS c_0,
g_0.e3 AS c_1 FROM pm1.g3 AS g_0 ORDER BY c_0
    AccessNode(5) output=[pm1.g2.e1, pm1.g2.e2] SELECT g_0.e1 AS c_0,
g_0.e2 AS c_1 FROM pm1.g2 AS g_0 ORDER BY c_0
```

These are the node properties:

Common

- Output Columns - what columns make up the tuples returned by this node
- Data Bytes Sent - how many data byte, not including messaging overhead, were sent by this query

- Planning Time - the amount of time in milliseconds spent planning the query

Relational

- Relational Node ID - matches the node ids seen in the debug log Node(id)
- Criteria - the boolean expression used for filtering
- Select Columns - the columns that define the projection
- Grouping Columns - the columns used for grouping
- Query - the source query
- Model Name - the model name
- Sharing ID - nodes sharing the same source results will have the same sharing id
- Dependent Join - if a dependent join is being used
- Join Strategy - the join strategy (Nested Loop, Sort Merge, Enhanced Sort, etc.)
- Join Type - the join type (Left Outer Join, Inner Join, Cross Join)
- Join Criteria - the join predicates
- Execution Plan - the nested execution plan
- Into Target - the insertion target
- Sort Columns - the columns for sorting
- Sort Mode - if the sort performs another function as well, such as distinct removal
- Rollup - if the group by has the rollup option
- Statistics - the processing statistics
- Cost Estimates - the cost/cardinality estimates including dependent join cost estimates
- Row Offset - the row offset expression
- Row Limit - the row limit expression
- With - the with clause
- Window Functions - the window functions being computed
- Table Function - the table function (XMLTABLE, OBJECTTABLE, TEXTTABLE, etc.)

XML

- Message
- Tag
- Namespace

- Data Column
- Namespace Declarations
- Optional Flag
- Default Value
- Recursion Direction
- Bindings
- Is Staging Flag
- Source In Memory Flag
- Condition
- Default Program
- Encoding
- Formatted Flag

Procedure

- Expression
- Result Set
- Program
- Variable
- Then
- Else

XML document model queries and procedure execution (including instead of triggers) use intermediate and final plan forms that include relational plans. Generally the structure of the xml/procedure plans will closely match their logical forms. It is the nested relational plans that will be of interest when analyzing performance issues.

14.9. QUERY PLANNER

14.9.1. Query Planner

For each sub-command in the user command one of the following sub-planners is used:

- Relational Planner
- Procedure Planner
- XML Planner

Each planner has three primary phases:

1. Generate canonical plan
2. Optimization
3. Plan to process converter - converts plan data structure into a processing form

14.9.2. Relational Planner

A relational processing plan is created by the optimizer after the logical plan is manipulated by a series of rules. The application of rules is determined both by the query structure and by the rules themselves. The node structure of the debug plan resembles that of the processing plan, but the node types more logically represent SQL operations.

User SQL statements after rewrite are converted into a canonical plan form. The canonical plan form most closely resembles the initial SQL structure. A SQL select query has the following possible clauses (all but SELECT are optional): WITH, SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. These clauses are logically executed in the following order:

- WITH (create common table expressions) - handled by a specialized PROJECT NODE
- FROM (read and join all data from tables) - SOURCE node for each from clause item, Join node (if >1 table)
- WHERE (filter rows) - SELECT node
- GROUP BY (group rows into collapsed rows) - GROUP node
- HAVING (filter grouped rows) - SELECT node
- SELECT (evaluate expressions and return only requested rows) - PROJECT node and DUP_REMOVE node (for SELECT DISTINCT)
- INTO - specialized PROJECT with a SOURCE child
- ORDER BY (sort rows) - SORT node
- LIMIT (limit result set to a certain range of results) - LIMIT node

For example, a SQL statement such as SELECT max(pm1.g1.e1) FROM pm1.g1 WHERE e2 = 1 creates a logical plan:

```
Project(groups=[anon_grp0], props={PROJECT_COLS=[anon_grp0.agg0 AS
expr1]})
  Group(groups=[anon_grp0], props={SYMBOL_MAP=
{anon_grp0.agg0=MAX(pm1.g1.e1)}})
    Select(groups=[pm1.g1], props={SELECT_CRITERIA=e2 = 1})
      Source(groups=[pm1.g1])
```

Here the Source corresponds to the FROM clause, the Select corresponds to the WHERE clause, the Group corresponds to the implied grouping to create the max aggregate, and the Project corresponds to the SELECT clause.

Note that the effect of grouping generates what is effectively an inline view, anon_grp0, to handle the projection of values created by the grouping.

- ACCESS - a source access or plan execution.

- DUP_REMOVE - removes duplicate rows
- JOIN - a join (LEFT OUTER, FULL OUTER, INNER, CROSS, SEMI, etc.)
- PROJECT - a projection of tuple values
- SELECT - a filtering of tuples
- SORT - an ordering operation, which may be inserted to process other operations such as joins
- SOURCE - any logical source of tuples including an inline view, a source access, XMLTABLE, etc.
- GROUP - a grouping operation
- SET_OP - a set operation (UNION/INTERSECT/EXCEPT)
- NULL - a source of no tuples
- TUPLE_LIMIT - row offset / limit

Each node has a set of applicable properties that are typically shown on the node.

- ATOMIC_REQUEST - The final form of a source request
- MODEL_ID - The metadata object for the target model/schema
- PROCEDURE_CRITERIA/PROCEDURE_INPUTS/PROCEDURE_DEFAULTS - Used in planning procedureal relational queries
- IS_MULTI_SOURCE - set to true when the node represents a multi-source access
- SOURCE_NAME - used to track the multi-source source name
- CONFORMED_SOURCES - tracks the set of conformed sources when the conformed extension metadata is used
- SUB_PLAN/SUB_PLANS - used in multi-source planning
- SET_OPERATION/USE_ALL - defines the set operation (UNION/INTERSECT/EXCEPT) and if all rows or distinct rows are used.

Join Properties

- JOIN_CRITERIA - all join predicates
- JOIN_TYPE - type of join (INNER, LEFT OUTER, etc.)
- JOIN_STRATEGY - the algorithm to use (nested loop, merge, etc.)
- LEFT_EXPRESSIONS - the expressions in equi-join predicates that originate from the left side of the join
- RIGHT_EXPRESSIONS - the expressions in equi-join predicates that originate from the right side of the join
- DEPENDENT_VALUE_SOURCE - set if a dependent join is used

- NON_EQUI_JOIN_CRITERIA - non-equi join predicates
- SORT_LEFT - if the left side needs sorted for join processing
- SORT_RIGHT - if the right side needs sorted for join processing
- IS_OPTIONAL - if the join is optional
- IS_LEFT_DISTINCT - if the left side is distinct with respect to the equi join predicates
- IS_RIGHT_DISTINCT - if the right side is distinct with respect to the equi join predicates
- IS_SEMI_DEP - if the dependent join represents a semi-join
- PRESERVE - if the preserve hint is preserving the join order

Project Properties

- PROJECT_COLS - the expressions projected
- INTO_GROUP - the group targeted if this is a select into or insert with a query expression
- HAS_WINDOW_FUNCTIONS - true if window functions are used
- CONSTRAINT - the constraint that must be met if the values are being projected into a group

Select Properties

- SELECT_CRITERIA - the filter
- IS_HAVING - if the filter is applied after grouping
- IS_PHANTOM - true if the node is marked for removal, but temporarily left in the plan.
- IS_TEMPORARY - inferred criteria that may not be used in the final plan
- IS_COPIED - if the criteria has already been processed by rule copy criteria
- IS_PUSHED - if the criteria is pushed as far as possible
- IS_DEPENDENT_SET - if the criteria is the filter of a dependent join

Sort Properties

- SORT_ORDER - the order by that defines the sort
- UNRELATED_SORT - if the ordering includes a value that is not being projected
- IS_DUP_REMOVAL - if the sort should also perform duplicate removal over the entire projection
- Source Properties - many source properties also become present on associated access nodes
- SYMBOL_MAP - the mapping from the columns above the source to the projected expressions. Also present on Group nodes
- PARTITION_INFO - the partitioning of the union branches

- VIRTUAL_COMMAND - if the source represents an view or inline view, the query that defined the view
- MAKE_DEP - hint information
- PROCESSOR_PLAN - the processor plan of a non-relational source (typically from the NESTED_COMMAND)
- NESTED_COMMAND - the non-relational command
- TABLE_FUNCTION - the table function (XMLTABLE, OBJECTTABLE, etc.) defining the source
- CORRELATED_REFERENCES - the correlated references for the nodes below the source
- MAKE_NOT_DEP - if make not dep is set
- INLINE_VIEW - If the source node represents an inline view
- NO_UNNEST - if the no_unnest hint is set
- MAKE_IND - if the make ind hint is set
- SOURCE_HINT - the source hint. See Federated Optimizations.
- ACCESS_PATTERNS - access patterns yet to be satisfied
- ACCESS_PATTERN_USED - satisfied access patterns
- REQUIRED_ACCESS_PATTERN_GROUPS - groups needed to satisfy the access patterns. Used in join planning.

Group Properties

- GROUP_COLS - the grouping columns
- ROLLUP - if the grouping includes a rollup

Tuple Limit Properties

- MAX_TUPLE_LIMIT - expression that evaluates to the max number of tuples generated
- OFFSET_TUPLE_COUNT - Expression that evaluates to the tuple offset of the starting tuple
- IS_IMPLICIT_LIMIT - if the limit is created by the rewriter as part of a subquery
- IS_NON_STRICT - if the unordered limit should not be enforced strictly optimization

General and Costing Properties

- OUTPUT_COLS - the output columns for the node. Is typically set after rule assign output elements.
- EST_SET_SIZE - represents the estimated set size this node would produce for a sibling node as the independent node in a dependent join scenario
- EST_DEP_CARDINALITY - value that represents the estimated cardinality (amount of rows) produced by this node as the dependent node in a dependent join scenario

- EST_DEP_JOIN_COST - value that represents the estimated cost of a dependent join (the join strategy for this could be Nested Loop or Merge)
- EST_JOIN_COST - value that represents the estimated cost of a merge join (the join strategy for this could be Nested Loop or Merge)
- EST_CARDINALITY - represents the estimated cardinality (amount of rows) produced by this node
- EST_COL_STATS - column statistics including number of null values, distinct value count,
- EST_SELECTIVITY - represents the selectivity of a criteria node

Relational optimization is based upon rule execution that evolves the initial plan into the execution plan. There are a set of pre-defined rules that are dynamically assembled into a rule stack for every query. The rule stack is assembled based on the contents of the user's query and the views/procedures accessed. For example, if there are no view layers, then rule Merge Virtual, which merges view layers together, is not needed and will not be added to the stack. This allows the rule stack to reflect the complexity of the query.

Logically the plan node data structure represents a tree of nodes where the source data comes up from the leaf nodes (typically Access nodes in the final plan), flows up through the tree and produces the user's results out the top. The nodes in the plan structure can have bidirectional links, dynamic properties, and allow any number of child nodes. Processing plans in contrast typically have fixed properties.

Plan rule manipulate the plan tree, fire other rules, and drive the optimization process. Each rule is designed to perform a narrow set of tasks. Some rules can be run multiple times. Some rules require a specific set of precursors to run properly.

- Access Pattern Validation - ensures that all access patterns have been satisfied
- Apply Security - applies row and column level security
- Assign Output Symbol - this rule walks top down through every node and calculates the output columns for each node. Columns that are not needed are dropped at every node, which is known as projection minimization. This is done by keeping track of both the columns needed to feed the parent node and also keeping track of columns that are “created” at a certain node.
- Calculate Cost - adds costing information to the plan
- Choose Dependent - this rule looks at each join node and determines whether the join should be made dependent and in which direction. Cardinality, the number of distinct values, and primary key information are used in several formulas to determine whether a dependent join is likely to be worthwhile. The dependent join differs in performance ideally because a fewer number of values will be returned from the dependent side. Also, we must consider the number of values passed from independent to dependent side. If that set is larger than the max number of values in an IN criteria on the dependent side, then we must break the query into a set of queries and combine their results. Executing each query in the connector has some overhead and that is taken into account. Without costing information a lot of common cases where the only criteria specified is on a non-unique (but strongly limiting) field are missed. A join is eligible to be dependent if:

  there is at least one equi-join criterion, i.e. tablea.col = tableb.col

  the join is not a full outer join and the dependent side of the join is on the inner side of the join

The join will be made dependent if one of the following conditions, listed in precedence order, holds:

There is an unsatisfied access pattern that can be satisfied with the dependent join criteria

The potential dependent side of the join is marked with an option makedep if costing was enabled, the estimated cost for the dependent join (possibly in each direction in the case of inner joins) is computed and compared to not performing the dependent join. If the costs were all determined (which requires all relevant table cardinality, column ndv, and possibly nnv values to be populated) the lowest is chosen.

If key metadata information indicates that the potential dependent side is not “small” and the other side is “not small” or the potential dependent side is the inner side of a left outer join.

Dependent join is the key optimization we use to efficiently process multi-source joins.

Instead of reading all of source A and all of source B and joining them on A.x = B.x, we read all of A then build a set of A.x that are passed as a criteria when querying B. In cases where A is small and B is large, this can drastically reduce the data retrieved from B, thus greatly speeding the overall query.

- Choose Join Strategy - choose the join strategy based upon the cost and attributes of the join.
- Clean Criteria - removes phantom criteria
- Collapse Source - takes all of the nodes below an access node and creates a SQL query representation
- Copy Criteria - this rule copies criteria over an equality criteria that is present in the criteria of a join. Since the equality defines an equivalence, this is a valid way to create a new criteria that may limit results on the other side of the join (especially in the case of a multi-source join).
- Decompose Join - this rule perfomes a partition-wise join optimization on joins of Federated Optimizations#Partitioned Union. The decision to decompose is based upon detecting that each side of the join is a partitioned union (note that non-ansi joins of more than 2 tables may cause the optimization to not detect the appropriate join). The rule currently only looks for situations where at most 1 partition matches from each side.
- Implement Join Strategy - adds necessary sort and other nodes to process the chosen join strategy
- Merge Criteria - combines select nodes and can convert subqueries to semi-joins
- Merge Virtual - removes view and inline view layers
- Place Access - places access nodes under source nodes. An access node represents the point at which everything below the access node gets pushed to the source or is a plan invocation. Later rules focus on either pushing under the access or pulling the access node up the tree to move more work down to the sources. This rule is also responsible for placing Federated Optimizations Access Patterns.
- Plan Joins - this rule attempts to find an optimal ordering of the joins performed in the plan, while ensuring that Federated Optimizations Access Patterns dependencies are met. This rule has three main steps. First it must determine an ordering of joins that satisfy the access patterns present. Second it will heuristically create joins that can be pushed to the source (if a set of joins are pushed to the source, we will not attempt to create an optimal ordering within that set. More than likely it will be sent to the source in the non-ANSI multi-join syntax and will

be optimized by the database). Third it will use costing information to determine the best left-linear ordering of joins performed in the processing engine. This third step will do an exhaustive search for 6 or less join sources and is heuristically driven by join selectivity for 7 or more sources.

- Plan Procedures - plans procedures that appear in procedural relational queries
- Plan Sorts - optimizations around sorting, such as combining sort operations or moving projection
- Plan Unions - reorders union children for more pushdown
- Plan Aggregates - performs aggregate decomposition over a join or union
- Push Limit - pushes the effect of a limit node further into the plan
- Push Non-Join Criteria - this rule will push predicates from the On Clause if it is not necessary for the correctness of the join.
- Push Select Criteria - pushed select nodes as far as possible through unions, joins, and views layers toward the access nodes. In most cases movement down the tree is good as this will filter rows earlier in the plan. We currently do not undo the decisions made by Push Select Criteria. However in situations where criteria cannot be evaluated by the source, this can lead to sub optimal plans.

One of the most important optimization related to pushing criteria is how the criteria will be pushed trough join. Consider the following plan tree that represents a subtree of the plan for the query "select ... from A inner join b on (A.x = B.x) where A.y = 3":

```
SELECT (B.y = 3)
       |
      JOIN - Inner Join on (A.x = B.x)
     /     \
  SRC (A)   SRC (B)
```

SELECT nodes represent criteria, and SRC stands for SOURCE.

It is always valid for inner join and cross joins to push (single source) criteria that are above the join, below the join. This allows for criteria originating in the user query to eventually be present in source queries below the joins. This result can be represented visually as:

```
JOIN - Inner Join on (A.x = B.x)
      /    \
     /   SELECT (B.y = 3)
    |        |
  SRC (A)   SRC (B)
```

The same optimization is valid for criteria specified against the outer side of an outer join.

```
SELECT (B.y = 3)
      |
     JOIN - Right Outer Join on (A.x = B.x)
    /     \
 SRC (A)   SRC (B)
```

This becomes:

```
    JOIN - Right Outer Join on (A.x = B.x)
    /    \
   /   SELECT (B.y = 3)
  |         |
SRC (A)   SRC (B)
```

However criteria specified against the inner side of an outer join needs special consideration. The above scenario with a left or full outer join is not the same.

```
SELECT (B.y = 3)
     |
    JOIN - Left Outer Join on (A.x = B.x)
   /     \
SRC (A)   SRC (B)
```

It becomes this

```
JOIN - Inner Join on (A.x = B.x)
      /    \
     /   SELECT (B.y = 3)
    |         |
  SRC (A)   SRC (B)
```

Since the criterion is not dependent upon the null values that may be populated from the inner side of the join, the criterion is eligible to be pushed below the join – but only if the join type is also changed to an inner join.

```
SELECT (B.y is null)
       |
      JOIN - Left Outer Join on (A.x = B.x)
     /     \
  SRC (A)   SRC (B)
```

This plan tree must have the criteria remain above the join, since the outer join may be introducing null values itself.

- Raise Access - this rule attempts to raise the Access nodes as far up the plan as possible. This is mostly done by looking at the source's capabilities and determining whether the operations can be achieved in the source or not.

- Raise Null - raises null nodes. Raising a null node removes the need to consider any part of the old plan that was below the null node.

- Remove Optional Joins - removes joins that are marked as or determined to be optional

- Substitute Expressions - used only when a function based index is present

- Validate Where All - ensures criteria is used when required by the source

As each relational sub plan is optimized, the plan will show what is being optimized and its canonical form:

```
OPTIMIZE:
```

```
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

----------------------------------------------------------------------
------
GENERATE CANONICAL:
SELECT e1 FROM (SELECT e1 FROM pm1.g1) AS x

CANONICAL PLAN:
Project(groups=[x], props={PROJECT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1,
SYMBOL_MAP={x.e1=e1}})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1]})
      Source(groups=[pm1.g1])
----------------------------------------------------------------------
------
```

With more complicated user queries, such as a procedure invocation or one containing subqueries, the sub plans may be nested within the overall plan. Each plan ends by showing the final processing plan:

```
OPTIMIZATION COMPLETE:
PROCESSOR PLAN:
AccessNode(0) output=[e1] SELECT g_0.e1 FROM pm1.g1 AS g_0
```

The effect of rules can be seen by the state of the plan tree before and after the rule fires. For example, the debug log below shows the application of rule merge virtual, which will remove the "x" inline view layer:

```
EXECUTING AssignOutputElements

AFTER:
Project(groups=[x], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Source(groups=[x], props={NESTED_COMMAND=SELECT e1 FROM pm1.g1,
SYMBOL_MAP={x.e1=e1}, OUTPUT_COLS=[e1]})
    Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
      Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema
name=pm1, nameInSource=null, uuid=3335, OUTPUT_COLS=[e1]})
        Source(groups=[pm1.g1], props={OUTPUT_COLS=[e1]})


==========================================================================
==
EXECUTING MergeVirtual

AFTER:
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=[e1]})
  Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema
name=pm1, nameInSource=null, uuid=3335, OUTPUT_COLS=[e1]})
    Source(groups=[pm1.g1])
```

Some important planning decisions are shown in the plan as they occur as an annotation. For example the snippet below shows that the access node could not be raised as the parent select node contained an unsupported subquery.

```
Project(groups=[pm1.g1], props={PROJECT_COLS=[e1], OUTPUT_COLS=null})
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */
(SELECT e1 FROM pm2.g1), OUTPUT_COLS=null})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema
name=pm1, nameInSource=null, uuid=3341, OUTPUT_COLS=null})
      Source(groups=[pm1.g1], props={OUTPUT_COLS=null})


================================================================================
==
EXECUTING RaiseAccess
LOW Relational Planner SubqueryIn is not supported by source pm1 - e1 IN
/*+ NO_UNNEST */ (SELECT e1 FROM pm2.g1) was not pushed

AFTER:
Project(groups=[pm1.g1])
  Select(groups=[pm1.g1], props={SELECT_CRITERIA=e1 IN /*+ NO_UNNEST */
(SELECT e1 FROM pm2.g1), OUTPUT_COLS=null})
    Access(groups=[pm1.g1], props={SOURCE_HINT=null, MODEL_ID=Schema
name=pm1, nameInSource=null, uuid=3341, OUTPUT_COLS=null})
      Source(groups=[pm1.g1])
```

Procedure Planner

The procedure planner is fairly simple. It converts the statements in the procedure into instructions in a program that will be run during processing. This is mostly a 1-to-1 mapping and very little optimization is performed.

The XML Planner creates an XML plan that is relatively close to the end result of the Procedure Planner – a program with instructions. Many of the instructions are even similar (while loop, execute SQL, etc). Additional instructions deal with producing the output result document (adding elements and attributes).

The XML planner does several types of planning (not necessarily in this order):

- Document selection - determine which tags of the virtual document should be excluded from the output document. This is done based on a combination of the model (which marks parts of the document excluded) and the query (which may specify a subset of columns to include in the SELECT clause).

- Criteria evaluation - breaks apart the user’s criteria, determine which result set the criteria should be applied to, and add that criteria to that result set query.

- Result set ordering - the query’s ORDER BY clause is broken up and the ORDER BY is applied to each result set as necessary

- Result set planning - ultimately, each result set is planned using the relational planner and taking into account all the impacts from the user's query. The planner will also look to automatically create staging tables and dependent joins based upon the mapping class hierarchy.

- Program generation - a set of instructions to produce the desired output document is produced, taking into account the final result set queries and the excluded parts of the document. Generally, this involves walking through the virtual document in document order, executing queries as necessary and emitting elements and attributes.

XML programs can also be recursive, which involves using the same document fragment for both the initial fragment and a set of repeated fragments (each a new query) until some termination criteria or limit is met.

XQuery is eligible for specific optimizations. Document projection is the most common optimization. It will be shown in the debug plan as an annotation. For example with the user query containing "xmltable('/a/b' passing doc columns x string path '@x', val string path '/.')", the debug plan would show a tree of the document that will effectively be used by the context and path XQuerys:

```
MEDIUM XQuery Planning Projection conditions met for /a/b - Document
projection will be used
childelement(Q{}a)
  childelement(Q{}b)
    attributeattribute(Q{}x)
      childtext()
    childtext()
```

APPENDIX A. BNF FOR SQL GRAMMAR

A.1. MAIN ENTRY POINTS

- callable statement
- ddl statement
- procedure body definition
- directly executable statement

A.2. RESERVED KEYWORDS

Keyword	Usage
ADD	add set option
ALL	standard aggregate function , function , query expression body , query term select clause , quantified comparison predicate
ALTER	alter , alter column options , alter options
AND	between predicate , boolean term
ANY	standard aggregate function , quantified comparison predicate
ARRAY_AGG	ordered aggregate function
AS	alter , array table , create procedure , option namespace , create table , create trigger , derived column , dynamic data statement , function , loop statement , xml namespace element , object table , select derived column , table subquery , text table , table name , with list element , xml serialize , xml table
ASC	sort specification
ATOMIC	compound statement , for each row trigger action
BEGIN	compound statement , for each row trigger action
BETWEEN	between predicate
BIGDECIMAL	data type
BIGINT	data type

Keyword	Usage
BIGINTEGER	data type
BLOB	data type , xml serialize
BOOLEAN	data type
BOTH	function
BREAK	branching statement
BY	group by clause , order by clause , window specification
BYTE	data type
CALL	callable statement , call statement
CASE	case expression , searched case expression
CAST	function
CHAR	function , data type
CLOB	data type , xml serialize
COLUMN	alter column options
CONSTRAINT	create table body
CONTINUE	branching statement
CONVERT	function
CREATE	create procedure , create foreign temp table , create table , create temporary table , create trigger , procedure body definition
CROSS	cross join
DATE	data type
DAY	function
DECIMAL	data type
DECLARE	declare statement

Keyword	Usage
DEFAULT	table element , xml namespace element , object table column , procedure parameter , xml table column
DELETE	alter , create trigger , delete statement
DESC	sort specification
DISTINCT	standard aggregate function , function , query expression body , query term , select clause
DOUBLE	data type
DROP	drop option , drop table
EACH	for each row trigger action
ELSE	case expression , if statement , searched case expression
END	case expression , compound statement , for each row trigger action , searched case expression
ERROR	raise error statement
ESCAPE	match predicate , text table
EXCEPT	query expression body
EXEC	dynamic data statement , call statement
EXECUTE	dynamic data statement , call statement
EXISTS	exists predicate
FALSE	non numeric literal
FETCH	fetch clause
FILTER	filter clause
FLOAT	data type
FOR	for each row trigger action , function , text aggregate function , xml table column
FOREIGN	alter options , create procedure , create foreign temp table , create table , foreign key

Keyword	Usage
FROM	delete statement , from clause , function
FULL	qualified table
FUNCTION	create procedure
GROUP	group by clause
HAVING	having clause
HOUR	function
IF	if statement
IMMEDIATE	dynamic data statement
IN	procedure parameter , in predicate
INNER	qualified table
INOUT	procedure parameter
INSERT	alter , create trigger , function , insert statement
INTEGER	data type
INTERSECT	query term
INTO	dynamic data statement , insert statement , into clause
IS	is null predicate
JOIN	cross join , qualified table
LANGUAGE	object table
LATERAL	table subquery
LEADING	function
LEAVE	branching statement
LEFT	function , qualified table
LIKE	match predicate

Keyword	Usage
LIKE_REGEX	like regex predicate
LIMIT	limit clause
LOCAL	create temporary table
LONG	data type
LOOP	loop statement
MAKEDEP	option clause , table primary
MAKENOTDEP	option clause , table primary
MERGE	insert statement
MINUTE	function
MONTH	function
NO	xml namespace element , text table column , text table
NOCACHE	option clause
NOT	between predicate , compound statement , table element , is null predicate , match predicate , boolean factor , procedure parameter , procedure result column , like regex predicate , in predicate , temporary table element
NULL	table element , is null predicate , non numeric literal , procedure parameter , procedure result column , temporary table element , xml query
OBJECT	data type
OF	alter , create trigger
OFFSET	limit clause
ON	alter , create foreign temp table , create trigger , loop statement , qualified table , xml query
ONLY	fetch clause
OPTION	option clause

Keyword	Usage
OPTIONS	alter options list , options clause
OR	boolean value expression
ORDER	order by clause
OUT	procedure parameter
OUTER	qualified table
OVER	window specification
PARAMETER	alter column options
PARTITION	window specification
PRIMARY	table element , create temporary table , primary key
PROCEDURE	alter , alter options , create procedure , procedure body definition
REAL	data type
REFERENCES	foreign key
RETURN	assignment statement , return statement , data statement
RETURNS	create procedure
RIGHT	function , qualified table
ROW	fetch clause , for each row trigger action , limit clause , text table
ROWS	fetch clause , limit clause
SECOND	function
SELECT	select clause
SET	add set option , option namespace , update statement
SHORT	data type

Keyword	Usage
SIMILAR	match predicate
SMALLINT	data type
SOME	standard aggregate function , quantified comparison predicate
SQLEXCEPTION	sql exception
SQLSTATE	sql exception
SQLWARNING	raise statement
STRING	dynamic data statement , data type , xml serialize
TABLE	alter options , create procedure , create foreign temp table , create table , create temporary table , drop table , query primary , table subquery
TEMPORARY	create foreign temp table , create temporary table
THEN	case expression , searched case expression
TIME	data type
TIMESTAMP	data type
TINYINT	data type
TO	match predicate
TRAILING	function
TRANSLATE	function
TRIGGER	alter , create trigger
TRUE	non numeric literal
UNION	cross join , query expression body
UNIQUE	other constraints , table element
UNKNOWN	non numeric literal
UPDATE	alter , create trigger , dynamic data statement , update statement

Keyword	Usage
USER	function
USING	dynamic data statement
VALUES	insert statement
VARBINARY	data type , xml serialize
VARCHAR	data type , xml serialize
VIRTUAL	alter options , create procedure , create table , procedure body definition
WHEN	case expression , searched case expression
WHERE	filter clause , where clause
WHILE	while statement
WITH	assignment statement , query expression , data statement
WITHOUT	assignment statement , data statement
XML	data type
XMLAGG	ordered aggregate function
XMLATTRIBUTES	xml attributes
XMLCOMMENT	function
XMLCONCAT	function
XMLELEMENT	xml element
XMLFOREST	xml forest
XMLNAMESPACES	xml namespaces
XMLPARSE	xml parse
XMLPI	function
XMLQUERY	xml query
XMLSERIALIZE	xml serialize

Keyword	Usage
XMLTABLE	xml table
YEAR	function

A.3. NON-RESERVED KEYWORDS

Keyword	Usage
ACCESSPATTERN	other constraints , non-reserved identifier
ARRAYTABLE	array table , non-reserved identifier
AUTO_INCREMENT	table element , non-reserved identifier
AVG	standard aggregate function , non-reserved identifier
CHAIN	sql exception , non-reserved identifier
COLUMNS	array table , non-reserved identifier , object table , text table , xml table
CONTENT	non-reserved identifier , xml parse , xml serialize
COUNT	standard aggregate function , non-reserved identifier
DELIMITER	non-reserved identifier , text aggregate function , text table
DENSE_RANK	analytic aggregate function , non-reserved identifier
DISABLED	alter , non-reserved identifier
DOCUMENT	non-reserved identifier , xml parse , xml serialize
EMPTY	non-reserved identifier , xml query
ENABLED	alter , non-reserved identifier
ENCODING	non-reserved identifier , text aggregate function , xml serialize
EVERY	standard aggregate function , non-reserved identifier

Keyword	Usage
EXCEPTION	compound statement , declare statement , non-reserved identifier
EXCLUDING	non-reserved identifier , xml serialize
EXTRACT	function , non-reserved identifier
FIRST	fetch clause , non-reserved identifier , sort specification
HEADER	non-reserved identifier , text aggregate function , text table
INCLUDING	non-reserved identifier , xml serialize
INDEX	other constraints , table element , non-reserved identifier
INSTEAD	alter , create trigger , non-reserved identifier
JSONARRAY_AGG	non-reserved identifier , ordered aggregate function
JSONOBJECT	json object , non-reserved identifier
KEY	table element , create temporary table , foreign key , non-reserved identifier , primary key
LAST	non-reserved identifier , sort specification
MAX	standard aggregate function , non-reserved identifier
MIN	standard aggregate function , non-reserved identifier
NAME	function , non-reserved identifier , xml element
NAMESPACE	option namespace , non-reserved identifier
NEXT	fetch clause , non-reserved identifier
NULLS	non-reserved identifier , sort specification
OBJECTTABLE	non-reserved identifier , object table
ORDINALITY	non-reserved identifier , xml table column

Keyword	Usage
PASSING	non-reserved identifier , object table , xml query , xml table
PATH	non-reserved identifier , xml table column
QUERYSTRING	non-reserved identifier , querystring function
QUOTE	non-reserved identifier , text aggregate function , text table
RAISE	non-reserved identifier , raise statement
RANK	analytic aggregate function , non-reserved identifier
RESULT	non-reserved identifier , procedure parameter
ROW_NUMBER	analytic aggregate function , non-reserved identifier
SELECTOR	non-reserved identifier , text table column , text table
SERIAL	non-reserved identifier , temporary table element
SKIP	non-reserved identifier , text table
SQL_TSI_DAY	time interval , non-reserved identifier
SQL_TSI_FRAC_SECOND	time interval , non-reserved identifier
SQL_TSI_HOUR	time interval , non-reserved identifier
SQL_TSI_MINUTE	time interval , non-reserved identifier
SQL_TSI_MONTH	time interval , non-reserved identifier
SQL_TSI_QUARTER	time interval , non-reserved identifier
SQL_TSI_SECOND	time interval , non-reserved identifier
SQL_TSI_WEEK	time interval , non-reserved identifier
SQL_TSI_YEAR	time interval , non-reserved identifier
STDDEV_POP	standard aggregate function , non-reserved identifier
STDDEV_SAMP	standard aggregate function , non-reserved identifier

Keyword	Usage
SUBSTRING	function , non-reserved identifier
SUM	standard aggregate function , non-reserved identifier
TEXTAGG	non-reserved identifier , text aggregate function
TEXTTABLE	non-reserved identifier , text table
TIMESTAMPADD	function , non-reserved identifier
TIMESTAMPDIFF	function , non-reserved identifier
TO_BYTES	function , non-reserved identifier
TO_CHARS	function , non-reserved identifier
TRIM	function , non-reserved identifier , text table column
VARIADIC	non-reserved identifier , procedure parameter
VAR_POP	standard aggregate function , non-reserved identifier
VAR_SAMP	standard aggregate function , non-reserved identifier
VERSION	non-reserved identifier , xml serialize
VIEW	alter , alter options , create table , non-reserved identifier
WELLFORMED	non-reserved identifier , xml parse
WIDTH	non-reserved identifier , text table column
XMLDECLARATION	non-reserved identifier , xml serialize

A.4. RESERVED KEYWORDS FOR FUTURE USE

ALLOCATE	ARE	ARRAY	ASENSITIVE	ASYMETRIC	AUTHORIZATION	BINARY	CALLED
CASCADED	CHARACTER	CHECK	CLOSE	COLLATE	COMMIT	CONNECT	CORRESPONDING

CRITERIA	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	CYCLE	DATALINK
DEALLOCATE	DEC	DEREF	DESCRIBE	DETERMINISTIC	DISCONNECT	DLNEWCOPY	DLPREVIOUSCOPY
DLURLCOMPLETE	DLURLCOMPLETEONLY	DLURLCOMPLETEWRITE	DLURLPATH	DLURLPATHONLY	DLURLPATHWRITE	DLURLSCHEME	DLURLSERVER
DLVALUE	DYNAMIC	ELEMENT	EXTERNAL	FREE	GET	GLOBAL	GRANT
HAS	HOLD	IDENTITY	IMPORT	INDICATOR	INPUT	INSENSITIVE	INT
INTERVAL	ISOLATION	LARGE	LOCALTIME	LOCALTIMESTAMP	MATCH	MEMBER	METHOD
MODIFIES	MODULE	MULTISET	NATIONAL	NATURAL	NCHAR	NCLOB	NEW
NONE	NUMERIC	OLD	OPEN	OUTPUT	OVERLAPS	PRECISION	PREPARE
RANGE	READS	RECURSIVE	REFERENCING	RELEASE	REVOKE	ROLLBACK	ROLLUP
SAVEPOINT	SCROLL	SEARCH	SENSITIVE	SESSION_USER	SPECIFIC	SPECIFICTYPE	SQL
START	STATIC	SUBMULTILIST	SYMETRIC	SYSTEM	SYSTEM_USER	TIMEZONE_HOUR	TIMEZONE_MINUTE
TRANSLATION	TREAT	VALUE	VARYING	WHENEVER	WINDOW	WITHIN	XMLBINARY
XMLCAST	XMLDOCUMENT	XMLEXISTS	XMLITERATE	XMLTEXT	XMLVALIDATE		

A.5. TOKENS

Name	Definition	Usage
all in group identifier	< identifier > < period > < star >	all in group

Name	Definition	Usage
binary string literal	"X" | "x" "\'" (< hexit > < hexit >)+ "\'"	non numeric literal
colon	":"	statement
comma	","	alter options list , column list , create procedure , typed element list , create table body , create temporary table , derived column list , sql exception named parameter list , expression list , from clause , function limit clause , object table , option clause , options clause , order by clause , data type , query expression , querystring function select clause , set clause list , in predicate , text aggregate function , text table , xml attributes , xml element , xml forest , xml namespaces , xml query , xml table
concat_op	"||"	common value expression
decimal numeric literal	(< digit >)* < period > < unsigned integer literal >	unsigned numeric literal
digit	["0"-"9"]	
dollar	"$"	unsigned value expression primary
eq	"="	assignment statement , callable statement , declare statement , named parameter list , comparison operator , set clause list
escaped function	"{" "fn"	unsigned value expression primary
escaped join	"{" "oj"	table reference
escaped type	"{" ("d" | "t" | "ts" | "b")	non numeric literal
approximate numeric literal	< digit > < period > < unsigned integer literal > ["e","E"] (< plus > | < minus >)? < unsigned integer literal >	unsigned numeric literal
ge	">="	comparison operator

Name	Definition	Usage
gt	">"	named parameter list , comparison operator
hexit	["a"-"f","A"-"F"] | < digit >	
identifier	< quoted_id > (< period > < quoted_id >)*	identifier , unsigned value expression primary
id_part	("@" | "#" | < letter >) (< letter > | "_" | < digit >)*	
lbrace	"{"	callable statement , match predicate
le	"<="	comparison operator
letter	["a"-"z","A"-"Z"] | ["\u0153"-"\ufffd"]	
lparen	"("	standard aggregate function , alter options list , analytic aggregate function , array table , callable statement , column list , other constraints , create procedure , create table body , create temporary table , filter clause , function , if statement , insert statement , json object , loop statement , object table , options clause , ordered aggreate function , data type , query primary , querystring function , in predicate , call statement , subquery , table subquery , table primary , text aggregate function , text table , unsigned value expression primary , while statement , window specification , with list element , xml attributes , xml element , xml forest , xml namespaces , xml parse , xml query , xml serialize , xml table
lsbrace	"["	unsigned value expression primary
lt	"<"	comparison operator
minus	"-"	plus or minus
ne	"<>"	comparison operator
ne2	"!="	comparison operator

Name	Definition	Usage
period	"."	
plus	"+"	plus or minus
qmark	"?"	callable statement , integer parameter , unsigned value expression primary
quoted_id	< id_part > | "\"" ("\"\"" | ~ ["\""])+ "\""	
rbrace	"}"	callable statement , match predicate , non numeric literal , table reference , unsigned value expression primary
rparen	")"	standard aggregate function , alter options list , analytic aggregate function , array table , callable statement , column list , other constraints , create procedure , create table body , create temporary table , filter clause , function , if statement , insert statement , json object , loop statement , object table , options clause , ordered aggregate function , data type , query primary , querystring function , in predicate , call statement , subquery , table subquery , table primary , text aggregate function , text table , unsigned value expression primary , while statement , window specification , with list element , xml attributes , xml element , xml forest , xml namespaces , xml parse , xml query , xml serialize , xml table
rsbrace	"]"	unsigned value expression primary
semicolon	";"	ddl statement , delimited statement
slash	"/"	star or slash
star	"*"	standard aggregate function , dynamic data statement , select clause , star or slash

Name	Definition	Usage
string literal	("N" | "E")? "\'" ("\'\'" | ~["\'"])* "\'"	string
unsigned integer literal	(< digit >)+	unsigned integer , unsigned numeric literal

A.6. PRODUCTION CROSS-REFERENCE

Name	Usage
add set option	alter options list
standard aggregate function	unsigned value expression primary
all in group	select sublist
alter	directly executable statement
alter column options	alter options
alter options list	alter column options , alter options
alter options	ddl statement
analytic aggregate function	unsigned value expression primary
array table	table primary
assignment statement	delimited statement
assignment statement operand	assignment statement , declare statement
between predicate	boolean primary
boolean primary	filter clause , boolean factor
branching statement	delimited statement
case expression	unsigned value expression primary
character	match predicate , text aggregate function , text table
column list	other constraints , create temporary table , foreign key , insert statement primary key , with list element

Name	Usage
common value expression	between predicate , boolean primary , comparison predicate , sql exception , match predicate , like regex predicate , in predicate , text table , unsigned value expression primary
comparison predicate	boolean primary
boolean term	boolean value expression
boolean value expression	condition
compound statement	statement
other constraints	create table body
table element	create table body
create procedure	ddl statement
typed element list	array table , dynamic data statement
create foreign temp table	directly executable statement
option namespace	ddl statement
create table	ddl statement
create table body	create foreign temp table , create table
create temporary table	directly executable statement
create trigger	ddl statement , directly executable statement
condition	expression , having clause , if statement , qualified table , searched case expression , where clause , while statement
cross join	joined table
declare statement	delimited statement
delete statement	assignment statement operand , directly executable statement
delimited statement	statement
derived column	derived column list , object table , querystring function , text aggregate function , xml attributes , xml query , xml table

Name	Usage
derived column list	json object , xml forest
drop option	alter options list
drop table	directly executable statement
dynamic data statement	data statement
raise error statement	delimited statement
sql exception	assignment statement operand , exception reference
exception reference	sql exception , raise statement
named parameter list	call statement
exists predicate	boolean primary
expression	standard aggregate function , assignment statement operand , case expression , derived column , dynamic data statement , raise error statement , named parameter list , expression list , function , object table column , ordered aggregate function , querystring function , return statement , searched case expression , select derived column , set clause list , sort key , unsigned value expression primary , xml table column , xml element , xml parse , xml serialize
expression list	callable statement , other constraints , function , group by clause , insert statement , call statement , window specification
fetch clause	limit clause
filter clause	function , unsigned value expression primary
for each row trigger action	alter , create trigger
foreign key	create table body
from clause	query
function	unsigned value expression primary
group by clause	query
having clause	query

Name	Usage
identifier	alter , alter column options , alter options , array table , assignment statement , branching statement , callable statement , column list , compound statement , table element , create procedure , typed element list , create foreign temp table , option namespace , create table , create table body , create temporary table , create trigger , declare statement , delete statement , derived column , drop option , drop table , dynamic data statement , exception reference , named parameter list , foreign key , function , insert statement , into clause , loop statement , xml namespace element , object table column , object table , option clause , option pair , procedure parameter , procedure result column , query primary , select derived column , set clause list , statement , call statement , table subquery , temporary table element , text aggregate function , text table column , text table , table name , update statement , with list element , xml table column , xml element , xml serialize , xml table
if statement	statement
insert statement	assignment statement operand , directly executable statement
integer parameter	fetch clause , limit clause
unsigned integer	dynamic data statement , integer parameter , data type , text table column , text table , unsigned value expression primary
time interval	function
into clause	query
is null predicate	boolean primary
joined table	table primary , table reference
json object	function
limit clause	query expression body
loop statement	statement
match predicate	boolean primary
xml namespace element	xml namespaces
non numeric literal	option pair , value expression primary

Name	Usage
non-reserved identifier	identifier , unsigned value expression primary
boolean factor	boolean term
object table column	object table
object table	table primary
comparison operator	comparison predicate , quantified comparison predicate
option clause	callable statement , delete statement , insert statement , query expression body , call statement , update statement
option pair	add set option , options clause
options clause	table element , create procedure , create table , create table body , procedure parameter , procedure result column
order by clause	function , ordered aggregate function , query expression body , text aggregate function , window specification
ordered aggregate function	unsigned value expression primary
data type	table element , create procedure , typed element list , declare statement , function , object table column , procedure parameter , procedure result column , temporary table element , text table column , xml table column
numeric value expression	common value expression
plus or minus	option pair , numeric value expression , value expression primary
primary key	create table body
procedure parameter	create procedure
procedure result column	create procedure
qualified table	joined table
query	query primary

Name	Usage
query expression	alter , assignment statement operand , create table , insert statement , loop statement , subquery , table subquery , directly executable statement , with list element
query expression body	query expression , query primary
query primary	query term
querystring function	function
query term	query expression body
raise statement	delimited statement
like regex predicate	boolean primary
return statement	delimited statement
searched case expression	unsigned value expression primary
select clause	query
select derived column	select sublist
select sublist	select clause
set clause list	dynamic data statement , update statement
in predicate	boolean primary
sort key	sort specification
sort specification	order by clause
data statement	delimited statement
statement	alter , compound statement , create procedure , for each row trigger action , if statement , loop statement , procedure body definition , while statement
call statement	assignment statement , subquery , table subquery , directly executable statement

Name	Usage
string	character , table element , option namespace , function , xml namespace element , non numeric literal , object table column , object table , procedure parameter , text table column , text table , xml table column , xml query , xml serialize , xml table
subquery	exists predicate , in predicate , quantified comparison predicate , unsigned value expression primary
quantified comparison predicate	boolean primary
table subquery	table primary
temporary table element	create temporary table
table primary	cross join , joined table
table reference	from clause , qualified table
text aggregate function	unsigned value expression primary
text table column	text table
text table	table primary
term	numeric value expression
star or slash	term
table name	table primary
unsigned numeric literal	option pair , value expression primary
unsigned value expression primary	value expression primary
update statement	assignment statement operand , directly executable statement
directly executable statement	data statement
value expression primary	array table , term
where clause	delete statement , query , update statement
while statement	statement
window specification	unsigned value expression primary

Name	Usage
with list element	query expression
xml attributes	xml element
xml table column	xml table
xml element	function
xml forest	function
xml namespaces	xml element , xml forest , xml query , xml table
xml parse	function
xml query	function
xml serialize	function
xml table	table primary

A.7. PRODUCTIONS

string ::=

- < string literal >

A string literal value. Use '' to escape ' in the string.

Example:

```
'a string'
```

```
'it''s a string'
```

reserved identifier ::=

- INSTEAD
- VIEW

-
  ```
  ENABLED
  ```

-
  ```
  DISABLED
  ```

-
  ```
  KEY
  ```

-
  ```
  SERIAL
  ```

-
  ```
  TEXTAGG
  ```

-
  ```
  COUNT
  ```

-
  ```
  ROW_NUMBER
  ```

-
  ```
  RANK
  ```

-
  ```
  DENSE_RANK
  ```

-
  ```
  SUM
  ```

-
  ```
  AVG
  ```

-
  ```
  MIN
  ```
-

```
MAX
```

-
  ```
  EVERY
  ```

-
  ```
  STDDEV_POP
  ```

-
  ```
  STDDEV_SAMP
  ```

-
  ```
  VAR_SAMP
  ```

-
  ```
  VAR_POP
  ```

-
  ```
  DOCUMENT
  ```

-
  ```
  CONTENT
  ```

-
  ```
  TRIM
  ```

-
  ```
  EMPTY
  ```

-
  ```
  ORDINALITY
  ```

-
  ```
  PATH
  ```

-
  ```
  FIRST
  ```

-
  ```
  LAST
  ```

-
  ```
  NEXT
  ```

-
  ```
  SUBSTRING
  ```

-
  ```
  EXTRACT
  ```

-
  ```
  TO_CHARS
  ```

-
  ```
  TO_BYTES
  ```

-
  ```
  TIMESTAMPADD
  ```

-
  ```
  TIMESTAMPDIFF
  ```

-
  ```
  QUERYSTRING
  ```

-
  ```
  NAMESPACE
  ```

-
  ```
  RESULT
  ```

-
  ```
  INDEX
  ```
-

```
ACCESSPATTERN
```

-
  ```
  AUTO_INCREMENT
  ```

-
  ```
  WELLFORMED
  ```

-
  ```
  SQL_TSI_FRAC_SECOND
  ```

-
  ```
  SQL_TSI_SECOND
  ```

-
  ```
  SQL_TSI_MINUTE
  ```

-
  ```
  SQL_TSI_HOUR
  ```

-
  ```
  SQL_TSI_DAY
  ```

-
  ```
  SQL_TSI_WEEK
  ```

-
  ```
  SQL_TSI_MONTH
  ```

-
  ```
  SQL_TSI_QUARTER
  ```

-
  ```
  SQL_TSI_YEAR
  ```

-
  ```
  TEXTTABLE
  ```

-
  ```
  ARRAYTABLE
  ```

-
  ```
  SELECTOR
  ```

-
  ```
  SKIP
  ```

-
  ```
  WIDTH
  ```

-
  ```
  PASSING
  ```

-
  ```
  NAME
  ```

-
  ```
  ENCODING
  ```

-
  ```
  COLUMNS
  ```

-
  ```
  DELIMITER
  ```

-
  ```
  QUOTE
  ```

-
  ```
  HEADER
  ```

-
  ```
  NULLS
  ```
-

```
OBJECTTABLE
```

-
  ```
  VERSION
  ```

-
  ```
  INCLUDING
  ```

-
  ```
  EXCLUDING
  ```

-
  ```
  XMLDECLARATION
  ```

-
  ```
  VARIADIC
  ```

-
  ```
  RAISE
  ```

-
  ```
  EXCEPTION
  ```

-
  ```
  CHAIN
  ```

-
  ```
  JSONARRAY_AGG
  ```

-
  ```
  JSONOBJECT
  ```

Allows non-reserved keywords to be parsed as identifiers

Example: SELECT **COUNT** FROM ...

identifier ::=

-

< identifier >

- < non-reserved identifier >

Partial or full name of a single entity.

Example:

```
tbl.col
```

```
"tbl"."col"
```

create trigger ::=

- CREATE TRIGGER ON < identifier > INSTEAD OF (INSERT | UPDATE | DELETE) AS < for each row trigger action >

Creates a trigger action on the given target.

Example:

```
CREATE TRIGGER ON vw INSTEAD OF INSERT AS FOR EACH ROW BEGIN ATOMIC ...
END
```

alter ::=

- ALTER ((VIEW < identifier > AS < query expression >) | (PROCEDURE < identifier > AS < statement >) | (TRIGGER ON < identifier > INSTEAD OF (INSERT | UPDATE | DELETE) ((AS < for each row trigger action >) | ENABLED | DISABLED)))

Alter the given target.

Example:

```
ALTER VIEW vw AS SELECT col FROM tbl
```

for each row trigger action ::=

- FOR EACH ROW ((BEGIN (ATOMIC)? (< statement >)* END) | < statement >)

Defines an action to perform on each row.

Example:

```
FOR EACH ROW BEGIN ATOMIC ... END
```

directly executable statement ::=

- < query expression >
- < call statement >
- < insert statement >
- < update statement >
- < delete statement >
- < drop table >
- < create temporary table >
- < create foreign temp table >
- < alter >
- < create trigger >

A statement that can be executed at runtime.

Example:

```
SELECT * FROM tbl
```

drop table ::=

- DROP TABLE < identifier >

Creates a trigger action on the given target.

Example:

```
CREATE TRIGGER ON vw INSTEAD OF INSERT AS FOR EACH ROW BEGIN ATOMIC ...
END
```

create temporary table ::=

- CREATE LOCAL TEMPORARY TABLE < identifier > < lparen > < temporary table element > (< comma > < temporary table element >)* (< comma > PRIMARY KEY < column list >)? < rparen >

Creates a temporary table.

Example:

```
CREATE LOCAL TEMPORARY TABLE tmp (col integer)
```

temporary table element ::=

- < identifier > (< data type > | SERIAL) (NOT NULL)?

Defines a temporary table column.

Example:

```
col string NOT NULL
```

raise error statement ::=

- ERROR < expression >

Raises an error with the given message.

Example:

```
ERROR 'something went wrong'
```

raise statement ::=

- RAISE (SQLWARNING)? < exception reference >

Raises an error or warning with the given message.

Example:

```
RAISE SQLEXCEPTION 'something went wrong'
```

exception reference ::=

- < identifier >
- < sql exception >

a reference to an exception

Example:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

sql exception ::=

- SQLEXCEPTION < common value expression > (SQLSTATE < common value expression > (< comma > < common value expression >)?)? (CHAIN < exception reference >)?

creates a sql exception or warning with the specified message, state, and code

Example:

```
SQLEXCEPTION 'something went wrong' SQLSTATE '00X', 2
```

statement ::=

- ((< identifier > < colon >)? (< loop statement > **|** < while statement > **|** < compound statement >))
- < if statement > **|** < delimited statement >

A procedure statement.

Example:

```
IF (x = 5) BEGIN ... END
```

delimited statement ::=

- (< assignment statement > | < data statement > | < raise error statement > | < raise statement > | < declare statement > | < branching statement > | < return statement >) < semicolon >

A procedure statement terminated by ;.

Example:

```
SELECT * FROM tbl;
```

compound statement ::=

- BEGIN ((NOT)? ATOMIC)? (< statement >)* (EXCEPTION < identifier > (< statement >)*)? END

A procedure statement block contained in BEGIN END.

Example:

```
BEGIN NOT ATOMIC ... END
```

branching statement ::=

- ((BREAK | CONTINUE) (< identifier >)?)

- (LEAVE < identifier >)

A procedure branching control statement, which typically specifies a label to return control to.

Example:

```
BREAK x
```

return statement ::=

-

```
RETURN ( < expression > )?
```

A return statement.

Example:

```
RETURN 1
```

while statement ::=

-
  ```
  WHILE < lparen > < condition > < rparen > < statement >
  ```

A procedure while statement that executes until its condition is false.

Example:

```
WHILE (var) BEGIN ... END
```

loop statement ::=

-
  ```
  LOOP ON < lparen > < query expression > < rparen > AS < identifier > < statement >
  ```

A procedure loop statement that executes over the given cursor.

Example:

```
IF (boolVal) BEGIN variables.x = 1 END ELSE BEGIN variables.x = 2 END
```

if statement ::=

-
  ```
  IF < lparen > < condition > < rparen > < statement > ( ELSE < statement > )?
  ```

A procedure loop statement that executes over the given cursor.

Example:

```
LOOP ON (SELECT * FROM tbl) AS x BEGIN ... END
```

declare statement ::=

-

```
DECLARE ( < data type > | EXCEPTION ) < identifier > ( < eq > < assignment statement operand > )?
```

A procedure declaration statement that creates a variable and optionally assigns a value.

Example:

```
DECLARE STRING x = 'a'
```

assignment statement ::=

-
  ```
  < identifier > < eq > ( < assignment statement operand > | ( < call statement > ( ( WITH | WITHOUT ) RETURN )? ) )
  ```

Assigns a variable a value in a procedure.

Example:

```
x = 'b'
```

assignment statement operand ::=

-
  ```
  < insert statement >
  ```

-
  ```
  < update statement >
  ```

-
  ```
  < delete statement >
  ```

-
  ```
  < expression >
  ```

-
  ```
  < query expression >
  ```

-
  ```
  < sql exception >
  ```

A value or command that can be used in an assignment.

NOTE

All assignments except for expression are deprecated.

data statement ::=

- (< directly executable statement > | < dynamic data statement >) ((WITH | WITHOUT) RETURN)?

A procedure statement that executes a SQL statement. An update statement can have its update count accessed via the ROWCOUNT variable.

procedure body definition ::=

- (CREATE (VIRTUAL)? PROCEDURE)? < statement >

Defines a procedure body on a Procedure metadata object.

Example:

```
CREATE VIRTUAL PROCEDURE BEGIN ... END
```

dynamic data statement ::=

- (EXECUTE | EXEC) (STRING | IMMEDIATE)? < expression > (AS < typed element list > (INTO < identifier >)?)? (USING < set clause list >)? (UPDATE (< unsigned integer > | < star >))?

A procedure statement that can execute arbitrary sql.

Example:

```
EXECUTE IMMEDIATE 'SELECT * FROM tbl' AS x STRING INTO #temp
```

set clause list ::=

- < identifier > < eq > < expression > (< comma > < identifier > < eq > < expression >)*

A list of value assignments.

Example:

```
col1 = 'x', col2 = 'y' ...
```

typed element list ::=

- < identifier > < data type > (< comma > < identifier > < data type >)*

A list of typed elements.

Example:

```
col1 string, col2 integer ...
```

callable statement ::=

- < lbrace > (< qmark > < eq >)? CALL < identifier > (< lparen > (< expression list >)? < rparen >)? < rbrace > (< option clause >)?

A callable statement defined using JDBC escape syntax.

Example:

```
{? = CALL proc}
```

call statement ::=

- ((EXEC | EXECUTE | CALL) < identifier > < lparen > (< named parameter list > | (< expression list >)?) < rparen >) (< option clause >)?

Executes the procedure with the given parameters.

Example:

```
CALL proc('a', 1)
```

named parameter list ::=

- (< identifier > < eq > (< gt >)? < expression > (< comma > < identifier > < eq > (< gt >)? < expression >)*)

A list of named parameters.

Example:

```
param1 => 'x', param2 => 1
```

insert statement ::=

-

```
( INSERT | MERGE ) INTO < identifier > ( < column list > )? ( ( VALUES < lparen > < expression list >
< rparen > ) | < query expression > ) ( < option clause > )?
```

Inserts values into the given target.

Example:

```
INSERT INTO tbl (col1, col2) VALUES ('a', 1)
```

expression list ::=

-
  ```
  < expression > ( < comma > < expression > )*
  ```

A list of expressions.

Example:

```
col1, 'a', ...
```

update statement ::=

-
  ```
  UPDATE < identifier > SET < set clause list > ( < where clause > )? ( < option clause > )?
  ```

Update values in the given target.

Example:

```
UPDATE tbl SET (col1 = 'a') WHERE col2 = 1
```

delete statement ::=

-
  ```
  DELETE FROM < identifier > ( < where clause > )? ( < option clause > )?
  ```

Delete rows from the given target.

Example:

```
DELETE FROM tbl WHERE col2 = 1
```

query expression ::=

-
  ```
  ( WITH < with list element > ( < comma > < with list element > )* )? < query expression body >
  ```

A declarative query for data.

Example:

```
SELECT * FROM tbl WHERE col2 = 1
```

with list element ::=

- < identifier > (< column list >)? AS < lparen > < query expression > < rparen >

A query expression for use in the enclosing query.

Example:

```
X (Y, Z) AS (SELECT 1, 2)
```

query expression body ::=

- < query term > ((UNION | EXCEPT) (ALL | DISTINCT)? < query term >)* (< order by clause >)? (< limit clause >)? (< option clause >)?

The body of a query expression, which can optionally be ordered and limited.

Example:

```
SELECT * FROM tbl ORDER BY col1 LIMIT 1
```

query term ::=

- < query primary > (INTERSECT (ALL | DISTINCT)? < query primary >)*

Used to establish INTERSECT precedence.

Example:

```
SELECT * FROM tbl
```

```
SELECT * FROM tbl1 INTERSECT SELECT * FROM tbl2
```

query primary ::=

- < query >

- (TABLE < identifier >)

- (< lparen > < query expression body > < rparen >)

A declarative source of rows.

Example:

```
TABLE tbl
```

```
SELECT * FROM tbl1
```

query ::=

- < select clause > (< into clause >)? (< from clause > (< where clause >)? (< group by clause >)? (< having clause >)?)?

A SELECT query.

Example:

```
SELECT col1, max(col2) FROM tbl GROUP BY col1
```

into clause ::=

- INTO < identifier >

Used to direct the query into a table.

NOTE

This is deprecated. Use INSERT INTO with a query expression instead.

Example:

```
INTO tbl
```

select clause ::=

- SELECT (ALL | DISTINCT)? (< star > | (< select sublist > (< comma > < select sublist >)*))

The columns returned by a query. Can optionally be distinct.

Example:

```
SELECT *
```

```
SELECT DISTINCT a, b, c
```

select sublist ::=

-
  ```
  < select derived column >
  ```

-
  ```
  < all in group >
  ```

An element in the select clause

Example:

```
tbl.*
```

```
tbl.col AS x
```

select derived column ::=

-
  ```
  ( < expression > ( ( AS )? < identifier > )? )
  ```

A select clause item that selects a single column.

NOTE

This is slightly different than a derived column in that the AS keyword is optional.

Example:

```
tbl.col AS x
```

derived column ::=

-
  ```
  ( < expression > ( AS < identifier > )? )
  ```

An optionally named expression.

Example:

```
tbl.col AS x
```

all in group ::=

- < all in group identifier >

A select sublist that can select all columns from the given group.

Example:

```
tbl.*
```

ordered aggreate function ::=

- (XMLAGG | ARRAY_AGG | JSONARRAY_AGG) < lparen > < expression > (< order by clause >)? < rparen >

An aggregate function that can optionally be ordered.

Example:

```
XMLAGG(col1) ORDER BY col2
```

```
ARRAY_AGG(col1)
```

text aggreate function ::=

- TEXTAGG < lparen > (FOR)? < derived column > (< comma > < derived column >)* (DELIMITER < character >)? (QUOTE < character >)? (HEADER)? (ENCODING < identifier >)? (< order by clause >)? < rparen >

An aggregate function for creating separated value clobs.

Example:

```
TEXTAGG (col1 as t1, col2 as t2 DELIMITER ',' HEADER)
```

standard aggregate function ::=

- (COUNT < lparen > < star > < rparen >)

-
  ```
  ( ( COUNT | SUM | AVG | MIN | MAX | EVERY | STDDEV_POP | STDDEV_SAMP | VAR_SAMP |
  VAR_POP | SOME | ANY ) < lparen > ( DISTINCT | ALL )? < expression > < rparen > )
  ```

A standard aggregate function.

Example:

```
COUNT(*)
```

analytic aggregate function ::=

-
  ```
  ( ROW_NUMBER | RANK | DENSE_RANK ) < lparen > < rparen >
  ```

An analytic aggregate function.

Example:

```
ROW_NUMBER()
```

filter clause ::=

-
  ```
  FILTER < lparen > WHERE < boolean primary > < rparen >
  ```

An aggregate filter clause applied prior to accumulating the value.

Example:

```
FILTER (WHERE col1='a')
```

from clause ::=

-
  ```
  FROM ( < table reference > ( < comma > < table reference > )* )
  ```

A query from clause containing a list of table references.

Example:

```
FROM a, b
```

```
FROM a right outer join b, c, d join e".</p>
```

table reference ::=

-
  ```
  ( < escaped join > < joined table > < rbrace > )
  ```

-
  ```
  < joined table >
  ```

An optionally escaped joined table.

Example:

```
a
```

```
a inner join b
```

joined table ::=

-
  ```
  < table primary > ( < cross join > | < qualified table > )*
  ```

A table or join.

Example:

```
a
```

```
a inner join b
```

cross join ::=

-
  ```
  ( ( CROSS | UNION ) JOIN < table primary > )
  ```

A cross join.

Example:

```
a CROSS JOIN b
```

qualified table ::=

-
  ```
  ( ( ( RIGHT ( OUTER )? ) | ( LEFT ( OUTER )? ) | ( FULL ( OUTER )? ) | INNER )? JOIN < table reference
  > ON < condition > )
  ```

An INNER or OUTER join.

Example:

```
a inner join b
```

table primary ::=

- (< text table > | < array table > | < xml table > | < object table > | < table name > | < table subquery > | (< lparen > < joined table > < rparen >)) (MAKEDEP | MAKENOTDEP)?

A single source of rows.

Example:

```
a
```

xml serialize ::=

- XMLSERIALIZE < lparen > (DOCUMENT | CONTENT)? < expression > (AS (STRING | VARCHAR | CLOB | VARBINARY | BLOB))? (ENCODING < identifier >)? (VERSION < string >)? ((INCLUDING | EXCLUDING) XMLDECLARATION)? < rparen >

Serializes an XML value.

Example:

```
XMLSERIALIZE(col1 AS CLOB)
```

array table ::=

- ARRAYTABLE < lparen > < value expression primary > COLUMNS < typed element list > < rparen > (AS)? < identifier >

The ARRAYTABLE table function creates tabular results from arrays. It can be used as a nested table reference.

Example:

```
ARRAYTABLE (col1 COLUMNS x STRING) AS y
```

text table ::=

-

```
TEXTTABLE < lparen > < common value expression > ( SELECTOR < string > )? COLUMNS < text table column > ( < comma > < text table column > )* ( NO ROW DELIMITER )? ( DELIMITER < character > )? ( ( ESCAPE < character > ) | ( QUOTE < character > ) )? ( HEADER ( < unsigned integer > )? )? ( SKIP < unsigned integer > )? < rparen > ( AS )? < identifier >
```

The TEXTTABLE table function creates tabular results from text. It can be used as a nested table reference.

Example:

```
TEXTTABLE (file COLUMNS x STRING) AS y
```

text table column ::=

-

```
< identifier > < data type > ( WIDTH < unsigned integer > ( NO TRIM )? )? ( SELECTOR < string > < unsigned integer > )?
```

A text table column.

Example:

```
x INTEGER WIDTH 6
```

xml query ::=

-

```
XMLQUERY < lparen > ( < xml namespaces > < comma > )? < string > ( PASSING < derived column > ( < comma > < derived column > )* )? ( ( NULL | EMPTY ) ON EMPTY )? < rparen >
```

Executes an XQuery to return an XML result.

Example:

```
XMLQUERY('<a>...</a>' PASSING doc)
```

object table ::=

-

```
OBJECTTABLE < lparen > ( LANGUAGE < string > )? < string > ( PASSING < derived column > ( < comma > < derived column > )* )? COLUMNS < object table column > ( < comma > < object table column > )* < rparen > ( AS )? < identifier >
```

Returns table results by processing a script.

Example:

```
OBJECTTABLE('z' PASSING val AS z COLUMNS col OBJECT 'teiid_row') AS X
```

object table column ::=

-
  ```
  < identifier > < data type > < string > ( DEFAULT < expression > )?
  ```

object table column.

Example:

```
y integer 'teiid_row_number'
```

xml table ::=

-
  ```
  XMLTABLE < lparen > ( < xml namespaces > < comma > )? < string > ( PASSING < derived column > ( < comma > < derived column > )* )? ( COLUMNS < xml table column > ( < comma > < xml table column > )* )? < rparen > ( AS )? < identifier >
  ```

Returns table results by processing an XQuery.

Example:

```
XMLTABLE('/a/b' PASSING doc COLUMNS col XML PATH '.') AS X
```

xml table column ::=

-
  ```
  < identifier > ( ( FOR ORDINALITY ) | ( < data type > ( DEFAULT < expression > )? ( PATH < string > )? ) )
  ```

XML table column.

Example:

```
y FOR ORDINALITY
```

unsigned integer ::=

-
  ```
  < unsigned integer literal >
  ```

An unsigned interger value.

Example:

```
12345
```

table subquery ::=

- (TABLE | LATERAL)? < lparen > (< query expression > | < call statement >) < rparen > (AS)? < identifier >

A table defined by a subquery.

Example:

```
(SELECT * FROM tbl) AS x
```

table name ::=

- (< identifier > ((AS)? < identifier >)?)

A table named in the FROM clause.

Example:

```
tbl AS x
```

where clause ::=

- WHERE < condition >

Specifies a search condition

Example:

```
WHERE x = 'a'
```

condition ::=

- < boolean value expression >

A boolean expression.

boolean value expression ::=

- < boolean term > (OR < boolean term >)*

An optionally ORed boolean expression.

boolean term ::=

-
  ```
  < boolean factor > ( AND < boolean factor > )*
  ```

An optional ANDed boolean factor.

boolean factor ::=

-
  ```
  ( NOT )? < boolean primary >
  ```

A boolean factor.

Example:

```
NOT x = 'a'
```

boolean primary ::=

-
  ```
  ( < common value expression > ( < between predicate > | < match predicate > | < like regex
  predicate > | < in predicate > | < is null predicate > | < quantified comparison predicate > | <
  comparison predicate > )? )
  ```

-
  ```
  < exists predicate >
  ```

A boolean predicate or simple expression.

Example:

```
col LIKE 'a%'
```

comparison operator ::=

-
  ```
  < eq >
  ```

-
  ```
  < ne >
  ```

-
  ```
  < ne2 >
  ```

-
  ```
  < lt >
  ```

-
  ```
  < le >
  ```

-
  ```
  < gt >
  ```

-
  ```
  < ge >
  ```

A comparison operator.

Example:

```
=
```

comparison predicate ::=

-
  ```
  < comparison operator > < common value expression >
  ```

A value comparison.

Example:

```
= 'a'
```

subquery ::=

-
  ```
  < lparen > ( < query expression > | < call statement > ) < rparen >
  ```

A subquery.

Example:

```
(SELECT * FROM tbl)
```

quantified comparison predicate ::=

-
  ```
  < comparison operator > ( ANY | SOME | ALL ) < subquery >
  ```

A subquery comparison.

Example:

```
= ANY (SELECT col FROM tbl)
```

match predicate ::=

- (NOT)? (LIKE | (SIMILAR TO)) < common value expression > (ESCAPE < character > | (< lbrace > ESCAPE < character > < rbrace >))?

Matches based upon a pattern.

Example:

```
LIKE 'a_'
```

like regex predicate ::=

- (NOT)? LIKE_REGEX < common value expression >

A regular expression match.

Example:

```
LIKE_REGEX 'a.*b'
```

character ::=

- < string >

A single character.

Example:

```
'a'
```

between predicate ::=

- (NOT)? BETWEEN < common value expression > AND < common value expression >

A comparison between two values.

Example:

```
BETWEEN 1 AND 5
```

is null predicate ::=

-
  ```
  IS ( NOT )? NULL
  ```

A null test.

Example:

```
IS NOT NULL
```

in predicate ::=

-
  ```
  ( NOT )? IN ( < subquery > | ( < lparen > < common value expression > ( < comma > < common value expression > )* < rparen > ) )
  ```

A comparison with multiple values.

Example:

```
IN (1, 5)
```

exists predicate ::=

-
  ```
  EXISTS < subquery >
  ```

A test if rows exist.

Example:

```
EXISTS (SELECT col FROM tbl)
```

group by clause ::=

-
  ```
  GROUP BY < expression list >
  ```

Defines the grouping columns

Example:

```
GROUP BY col1, col2
```

having clause ::=

- HAVING < condition >

Search condition applied after grouping.

Example:

```
HAVING max(col1) = 5
```

order by clause ::=

- ORDER BY < sort specification > (< comma > < sort specification >)*

Specifies row ordering.

Example:

```
ORDER BY x, y DESC
```

sort specification ::=

- < sort key > (ASC | DESC)? (NULLS (FIRST | LAST))?

Defines how to sort on a particular expression

Example:

```
col1 NULLS FIRST
```

sort key ::=

- < expression >

A sort expression.

Example:

```
col1
```

integer parameter ::=

-
  ```
  < unsigned integer >
  ```

-
  ```
  < qmark >
  ```

A literal integer or parameter reference to an integer.

Example:

```
?
```

limit clause ::=

-
  ```
  ( LIMIT < integer parameter > ( < comma > < integer parameter > )? )
  ```

-
  ```
  ( OFFSET < integer parameter > ( ROW | ROWS ) ( < fetch clause > )? )
  ```

-
  ```
  < fetch clause >
  ```

Limits and/or offsets the resultant rows.

Example:

```
LIMIT 2
```

fetch clause ::=

-
  ```
  FETCH ( FIRST | NEXT ) ( < integer parameter > )? ( ROW | ROWS ) ONLY
  ```

ANSI limit.

Example:

```
FETCH FIRST 1 ROWS ONLY
```

option clause ::=

-

```
OPTION ( MAKEDEP < identifier > ( < comma > < identifier > )* | MAKENOTDEP < identifier > ( <
comma > < identifier > )* | NOCACHE ( < identifier > ( < comma > < identifier > )* )? )*
```

Specifies query options.

Example:

```
OPTION MAKEDEP tbl
```

expression ::=

-
  ```
  < condition >
  ```

A value.

Example:

```
col1
```

common value expression ::=

-
  ```
  ( < numeric value expression > ( < concat_op > < numeric value expression > )* )
  ```

Establishes the precedence of concat.

Example:

```
'a' || 'b'
```

numeric value expression ::=

-
  ```
  ( < term > ( < plus or minus > < term > )* )
  ```

Example:

```
1 + 2
```

plus or minus ::=

-
  ```
  < plus >
  ```

-

```
< minus >
```

The + or - operator.

Example:

```
+
```

term ::=

-
  ```
  ( < value expression primary > ( < star or slash > < value expression primary > )* )
  ```

A numeric term

Example:

```
1 * 2
```

star or slash ::=

-
  ```
  < star >
  ```

-
  ```
  < slash >
  ```

The * or / operator.

Example:

```
/
```

value expression primary ::=

-
  ```
  < non numeric literal >
  ```

-
  ```
  ( < plus or minus > )? ( < unsigned numeric literal > | < unsigned value expression primary > )
  ```

A simple value expression.

Example:

```
+col1
```

unsigned value expression primary ::=

- < qmark >
- (< dollar > < unsigned integer >)
- (< escaped function > < function > < rbrace >)
- ((< text aggregate function > **|** < standard aggregate function > **|** < ordered aggregate function >) (< filter clause >)? (< window specification >)?)
- (< analytic aggregate function > (< filter clause >)? < window specification >)
- (< function > (< window specification >)?)
- ((< identifier > **|** < non-reserved identifier >) (< lsbrace > < common value expression > < rsbrace >)?)
- < subquery >
- (< lparen > < expression > < rparen > (< lsbrace > < common value expression > < rsbrace >)?)
- < searched case expression >
- < case expression >

An unsigned simple value expression.

Example:

```
col1
```

window specification ::=

- OVER < lparen > (PARTITION BY < expression list >)? (< order by clause >)? < rparen >

The window specification for an analytical or windowed aggregate function.

Example:

```
OVER (PARTION BY col1)
```

case expression ::=

- CASE < expression > (WHEN < expression > THEN < expression >)+ (ELSE < expression >)? END

If/then/else chain using a common search predicand.

Example:

```
CASE col1 WHEN 'a' THEN 1 ELSE 2
```

searched case expression ::=

- CASE (WHEN < condition > THEN < expression >)+ (ELSE < expression >)? END

If/then/else chain using multiple search conditions.

Example:

```
CASE WHEN x = 'a' THEN 1 WHEN y = 'b' THEN 2
```

function ::=

- (CONVERT < lparen > < expression > < comma > < data type > < rparen >)

- (CAST < lparen > < expression > AS < data type > < rparen >)

-
  ```
  ( SUBSTRING < lparen > < expression > ( ( FROM < expression > ( FOR < expression > )? ) | ( < comma > < expression list > ) ) < rparen > )
  ```

-
  ```
  ( EXTRACT < lparen > ( YEAR | MONTH | DAY | HOUR | MINUTE | SECOND ) FROM < expression > < rparen > )
  ```

-
  ```
  ( TRIM < lparen > ( ( ( ( LEADING | TRAILING | BOTH ) ( < expression > )? ) | < expression > ) FROM )? < expression > < rparen > )
  ```

-
  ```
  ( ( TO_CHARS | TO_BYTES ) < lparen > < expression > < comma > < string > < rparen > )
  ```

-
  ```
  ( ( TIMESTAMPADD | TIMESTAMPDIFF ) < lparen > < time interval > < comma > < expression > < comma > < expression > < rparen > )
  ```

-
  ```
  < querystring function >
  ```

-
  ```
  ( ( LEFT | RIGHT | CHAR | USER | YEAR | MONTH | HOUR | MINUTE | SECOND | XMLCONCAT | XMLCOMMENT ) < lparen > ( < expression list > )? < rparen > )
  ```

-
  ```
  ( ( TRANSLATE | INSERT ) < lparen > ( < expression list > )? < rparen > )
  ```

-
  ```
  < xml parse >
  ```

-
  ```
  < xml element >
  ```

-
  ```
  ( XMLPI < lparen > ( ( NAME )? < identifier > ) ( < comma > < expression > )? < rparen > )
  ```

-

```
< xml forest >
```

-
  ```
  < json object >
  ```

-
  ```
  < xml serialize >
  ```

-
  ```
  < xml query >
  ```

-
  ```
  ( < identifier > < lparen > ( ALL | DISTINCT )? ( < expression list > )? ( < order by clause > )? <
  rparen > ( < filter clause > )? )
  ```

Calls a scalar function.

Example:

```
func('1', col1)
```

xml parse ::=

-
  ```
  XMLPARSE < lparen > ( DOCUMENT | CONTENT ) < expression > ( WELLFORMED )? < rparen >
  ```

Parses the given value as XML.

Example:

```
XMLPARSE(DOCUMENT doc WELLFORMED)
```

querystring function ::=

-
  ```
  QUERYSTRING < lparen > < expression > ( < comma > < derived column > )* < rparen >
  ```

Produces a URL query string from the given arguments.

Example:

```
QUERYSTRING(col1 AS opt, col2 AS val)
```

xml element ::=

- XMLELEMENT < lparen > ((NAME)? < identifier >) (< comma > < xml namespaces >)? (< comma > < xml attributes >)? (< comma > < expression >)* < rparen >

Creates an XML element.

Example:

```
XMLELEMENT(NAME "root", child)
```

xml attributes ::=

- XMLATTRIBUTES < lparen > < derived column > (< comma > < derived column >)* < rparen >

Creates attributes for the containing element.

Example:

```
XMLATTRIBUTES(col1 AS attr1, col2 AS attr2)
```

json object ::=

- JSONOBJECT < lparen > < derived column list > < rparen >

Produces a JSON object containing name value pairs.

Example:

```
JSONOBJECT(col1 AS val1, col2 AS val2)
```

derived column list ::=

- < derived column > (< comma > < derived column >)*

a list of name value pairs

Example:

```
col1 AS val1, col2 AS val2
```

xml forest ::=

-

XMLFOREST < lparen > (< xml namespaces > < comma >)? < derived column list > < rparen >

Produces an element for each derived column.

Example:

```
XMLFOREST(col1 AS ELEM1, col2 AS ELEM2)
```

xml namespaces ::=

- XMLNAMESPACES < lparen > < xml namespace element > (< comma > < xml namespace element >)* < rparen >

Defines XML namespace URI/prefix combinations

Example:

```
XMLNAMESPACES('http://foo' AS foo)
```

xml namespace element ::=

- (< string > AS < identifier >)

- (NO DEFAULT)

- (DEFAULT < string >)

An xml namespace

Example:

```
NO DEFAULT
```

data type ::=

- (STRING (< lparen > < unsigned integer > < rparen >)?)

-

```
( VARCHAR ( < lparen > < unsigned integer > < rparen > )? )
```

-
  ```
  BOOLEAN
  ```

-
  ```
  BYTE
  ```

-
  ```
  TINYINT
  ```

-
  ```
  SHORT
  ```

-
  ```
  SMALLINT
  ```

-
  ```
  ( CHAR ( < lparen > < unsigned integer > < rparen > )? )
  ```

-
  ```
  INTEGER
  ```

-
  ```
  LONG
  ```

-
  ```
  BIGINT
  ```

-
  ```
  ( BIGINTEGER ( < lparen > < unsigned integer > < rparen > )? )
  ```

-
  ```
  FLOAT
  ```

-
  ```
  REAL
  ```

-
  ```
  DOUBLE
  ```

-
  ```
  ( BIGDECIMAL ( < lparen > < unsigned integer > ( < comma > < unsigned integer > )? < rparen > )? )
  ```

-
  ```
  ( DECIMAL ( < lparen > < unsigned integer > ( < comma > < unsigned integer > )? < rparen > )? )
  ```

-
  ```
  DATE
  ```

-
  ```
  TIME
  ```

-
  ```
  TIMESTAMP
  ```

-
  ```
  OBJECT
  ```

-
  ```
  ( BLOB ( < lparen > < unsigned integer > < rparen > )? )
  ```

-
  ```
  ( CLOB ( < lparen > < unsigned integer > < rparen > )? )
  ```

-
  ```
  ( VARBINARY ( < lparen > < unsigned integer > < rparen > )? )
  ```

-
  ```
  XML
  ```

A data type.

Example:

```
STRING
```

time interval ::=

-
  ```
  SQL_TSI_FRAC_SECOND
  ```

-
  ```
  SQL_TSI_SECOND
  ```

-
  ```
  SQL_TSI_MINUTE
  ```

-
  ```
  SQL_TSI_HOUR
  ```

-
  ```
  SQL_TSI_DAY
  ```

-
  ```
  SQL_TSI_WEEK
  ```

-
  ```
  SQL_TSI_MONTH
  ```

-
  ```
  SQL_TSI_QUARTER
  ```

-
  ```
  SQL_TSI_YEAR
  ```

A time interval keyword.

Example:

```
SQL_TSI_HOUR
```

non numeric literal ::=

-
  ```
  < string >
  ```
-

```
< binary string literal >
```

-
  ```
  FALSE
  ```

-
  ```
  TRUE
  ```

-
  ```
  UNKNOWN
  ```

-
  ```
  NULL
  ```

-
  ```
  ( < escaped type > < string > < rbrace > )
  ```

An escaped or simple non numeric literal.

Example:

```
'a'
```

unsigned numeric literal ::=

-
  ```
  < unsigned integer literal >
  ```

-
  ```
  < approximate numeric literal >
  ```

-
  ```
  < decimal numeric literal >
  ```

An unsigned numeric literal value.

Example:

```
1.234
```

ddl statement ::=

- (< create table > | < create procedure > | < option namespace > | < alter options > | < create trigger >) (< semicolon >)?

A data definition statement.

Example:

```
CREATE FOREIGN TABLE X (Y STRING)
```

option namespace ::=

- SET NAMESPACE < string > AS < identifier >

A namespace used to shorten the full name of an option key.

Example:

```
SET NAMESPACE 'http://foo' AS foo
```

create procedure ::=

- CREATE (VIRTUAL | FOREIGN)? (PROCEDURE | FUNCTION) (< identifier > < lparen > (< procedure parameter > (< comma > < procedure parameter >)*)? < rparen > (RETURNS (((TABLE)? < lparen > < procedure result column > (< comma > < procedure result column >)* < rparen >) | < data type >))? (< options clause >)? (AS < statement >)?)

Defines a procedure or function invocation.

Example:

```
CREATE FOREIGN PROCEDURE proc (param STRING) RETURNS STRING
```

procedure parameter ::=

- (IN | OUT | INOUT | VARIADIC)? < identifier > < data type > (NOT NULL)? (RESULT)? (DEFAULT < string >)? (< options clause >)?

A procedure or function parameter

Example:

```
OUT x INTEGER
```

procedure result column ::=

- < identifier > < data type > (NOT NULL)? (< options clause >)?

A procedure result column.

Example:

```
x INTEGER
```

create table ::=

- CREATE (FOREIGN TABLE | (VIRTUAL)? VIEW) < identifier > (< create table body > | (< options clause >)?) (AS < query expression >)?

Defines a table or view.

Example:

```
CREATE VIEW vw AS SELECT 1
```

create foreign temp table ::=

- CREATE FOREIGN TEMPORARY TABLE < identifier > < create table body > ON < identifier >

Defines a foreign temp table

Example:

```
CREATE FOREIGN TEMPORARY TABLE t (x string) ON z
```

create table body ::=

- (< lparen > < table element > (< comma > < table element >)* (< comma > (CONSTRAINT < identifier >)? (< primary key > | < other constraints > | < foreign key >) (< options clause >)?)* < rparen >)? (< options clause >)?

Defines a table.

Example:

```
(x string) OPTIONS (CARDINALITY 100)
```

foreign key ::=

-

```
FOREIGN KEY < column list > REFERENCES < identifier > ( < column list > )?
```

Defines the foreign key referential constraint.

Example:

```
FOREIGN KEY (a, b) REFERENCES tbl (x, y)
```

primary key ::=

-
  ```
  PRIMARY KEY < column list >
  ```

Defines the primary key.

Example:

```
PRIMARY KEY (a, b)
```

other constraints ::=

-
  ```
  ( ( UNIQUE | ACCESSPATTERN ) < column list > )
  ```

-
  ```
  ( INDEX < lparen > < expression list > < rparen > )
  ```

Defines ACCESSPATTERN and UNIQUE constraints and INDEXes.

Example:

```
UNIQUE (a)
```

column list ::=

-
  ```
  < lparen > < identifier > ( < comma > < identifier > )* < rparen >
  ```

A list of column names.

Example:

```
(a, b)
```

table element ::=

- < identifier > < data type > (NOT NULL)? (AUTO_INCREMENT)? ((PRIMARY KEY) | ((UNIQUE)? (INDEX)?)) (DEFAULT < string >)? (< options clause >)?

Defines a table column.

Example:

```
x INTEGER NOT NULL
```

options clause ::=

- OPTIONS < lparen > < option pair > (< comma > < option pair >)* < rparen >

A list of statement options.

Example:

```
OPTIONS ('x' 'y', 'a' 'b')
```

option pair ::=

- < identifier > (< non numeric literal > | (< plus or minus >)? < unsigned numeric literal >)

An option key/value pair.

Example:

```
'key' 'value'
```

alter options ::=

- ALTER (VIRTUAL | FOREIGN)? (TABLE | VIEW | PROCEDURE) < identifier > (< alter options list > | < alter column options >)

alters options of tables/procedure

Example:

```
ALTER FOREIGN TABLE foo OPTIONS (ADD cardinality 100)
```

alter options list ::=

-

```
OPTIONS < lparen > ( < add set option > | < drop option > ) ( < comma > ( < add set option > | < drop option > ) )* < rparen >
```

a list of alterations to options

Example:

```
OPTIONS (ADD updatable true)
```

drop option ::=

-
  ```
  DROP < identifier >
  ```

drop option

Example:

```
DROP updatable
```

add set option ::=

-
  ```
  ( ADD | SET ) < option pair >
  ```

add or set an option pair

Example:

```
ADD updatable true
```

alter column options ::=

-
  ```
  ALTER ( COLUMN | PARAMETER )? < identifier > < alter options list >
  ```

alters a set of column options

Example:

```
ALTER COLUMN bar OPTIONS (ADD updatable true)
```

APPENDIX B. DASHBOARD BUILDER (TECHNOLOGY PREVIEW)

B.1. JBOSS DASHBOARD BUILDER

JBoss Dashboard Builder is an open source dashboard and reporting tool that allows:

- Visual configuration and personalization of dashboards.
- Graphical representation of KPIs (Key Performance Indicators).
- Definition of interactive report tables.
- Filtering and search, both in-memory or database based.
- Process execution metrics dashboards.
- Data extraction from external systems, through different protocols.
- Access control for different user profiles to different levels of information.

B.2. TECHNOLOGY PREVIEW

WARNING

Technology preview features provide early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. However, these features are not fully supported under Subscription Level Agreements, may not be functionally complete, and are not intended for production use. As Red Hat considers making future iterations of technology preview features generally available, we will attempt to resolve any issues that customers experience when using these features. During the development of a technology preview feature, additional components may become available to the public for testing. Because technology preview features are still under development, Red Hat cannot guarantee the stability of such features. As a result, if you are using technology preview features, you may not be able to seamlessly upgrade to subsequent releases of that feature. While Red Hat intends to fully support technology preview features in future releases, we may discover that a feature does not meet the standards for enterprise viability. If this happens, we cannot guarantee that technology preview features will be released in a supported manner. Some technology preview features may only be available for specific hardware architectures.

NOTE

Red Hat JBoss support will provide commercially reasonable efforts to resolve any reported issues that customers experience when using these features.

B.3. LOG IN TO JBOSS DASHBOARD BUILDER

Prerequisites

- Red Hat JBoss Data Virtualization must be installed and running.
- You must have a JBoss Dashboard Builder user account.

Procedure B.1. Log in to the JBoss Dashboard Builder

1. **Navigate to JBoss Dashboard Builder**
   Navigate to JBoss Dashboard Builder in your web browser. The default location is http://localhost:8080/dashboard.

2. **Log in to JBoss Dashboard Builder**
   Enter the **`Username`** and **`Password`** of a valid JBoss Dashboard Builder user.

B.4. ADDING A JBOSS DASHBOARD BUILDER USER

A JBoss Dashboard Builder user is added in the same way as a JBoss Data Virtualization user.

Two roles are provided for setting JBoss Dashboard Builder permissions:

- **`user`** - a user has permission to view the dashboard
- **`admin`** - a user has permission to modify the dashboard

IMPORTANT

If a JBoss Dashboard Builder user wants their JBoss Data Virtualization permissions applied to the data they are accessing, then the external datasource defined in JBoss Dashboard Builder must use local connection and set PassthroughAuthentication to **`true`** on the URL; otherwise, the default username and password defined for the datasource are used.

Example B.1. The PassthroughAuthentication property set on the connection URL

```
jdbc:teiid:VDBName;PassthroughAuthentication="true"
```

APPENDIX C. REVISION HISTORY

Revision 6.2.0-23	**Mon Feb 22 2016**	**David Le Sage**

Updates for 6.2