Teiid - Scalable Information Integration

1

# Teiid Reference Documentation

7.0

# Preface

Teiid offers a highly scalable and high performance solution to information integration. By allowing integrated and enriched data to be consumed relationally or as XML over multiple protocols, Teiid simplifies data access for developers and consuming applications.

Commercial development support, production support, and training for Teiid is available through JBoss Inc. Teiid is a Professional Open Source project and a critical component of the JBoss Enterprise Data Services Platform.

# SQL Support

Teiid supports SQL for issuing queries and for defining view transformations; see also *Procedure Language* for how SQL is used in virtual procedures and update procedures.

Teiid provides nearly all of the functionality of SQL-92 DML. SQL-99 and later features are constantly being added based upon community need. The following does not attempt to cover SQL exhaustively, but rather highlights SQL's usage within Teiid. See the *grammar* for the exact form of SQL accepted by Teiid.

## 1.1. 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, or 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 it's be escaped with an additional double quote. e.g. "some "" id"
- Because different data sources organize tables in different ways, some prepending catalog or schema or user information, Teiid 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. e.g. a table with the fully-qualified name vdbname."sourcescema.sourcetable" would match the partial name sourcetable.

- Columns, schemas, and aliases identifiers cannot contain a dot.

- Identifiers, even when quoted, are not case-sensitive in Teiid.

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. Partially- 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.

## 1.2. 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.

Teiid supports the following types of expressions:

- *Column identifiers*
- *Literals*
- *Scalar functions*
- *Aggregate functions*
- *Case and searched case*
- *Scalar subqueries*
- *Parameter references*

### 1.2.1. 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 was defined in the *Identifiers* section above.

## 1.2.2. Literals

Literal values represent fixed values. These can any of the 'standard' *data 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 double
- `true`
- `false`
- `'\u0027'` - unicode character

## 1.2.3. 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.

Teiid supports the following aggregate functions:

- COUNT(*) – count the number of values (including nulls and duplicates) in a group
- COUNT(expression) – count the number of values (excluding nulls) in a group
- SUM(expression) – sum of the values (excluding nulls) in a group

- AVG(expression) – average of the values (excluding nulls) in a group
- MIN(expression) – minimum value in a group (excluding null)
- MAX(expression) – maximum value in a group (excluding null)
- XMLAGG(xml expression *[ORDER BY ...]*) – xml concatination of all xml expressions in a group (excluding null)

Syntax Rules:

- Some aggregate functions may contain a 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 may only be used in the HAVING or SELECT clauses and may not be nested within another aggregate function.
- Aggregate functions may be nested inside other functions.

For more information on aggregates, see the sections on GROUP BY or HAVING.

## 1.2.4. Case and searched case

Teiid 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.

## 1.2.5. 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, see the *Subqueries* section below.

### 1.2.6. Parameter references

Parameters are specified using a '?' symbol. Parameters may only be used with PreparedStatement or CallableStatements in JDBC. Each parameter is linked to a value specified by 1-based index in the JDBC API.

## 1.3. Criteria

Criteria are of two basic forms:

- Predicates that evaluate to true or false
- Logical criteria that combine predicates (AND, OR, NOT)

Syntax Rules:

- expression (=|<>|!=|<|>|<=|>=) (expression|((ANY|ALL|SOME) subquery))
- expression [NOT] IS NULL
- expression [NOT] IN (expression[,expression]*)|subquery
- expression [NOT] LIKE expression [ESCAPE char]
- EXISTS(subquery)
- expression BETWEEN minExpression AND maxExpression
- criteria AND|OR criteria
- NOT criteria
- Criteria may be nested using parenthesis.

Some examples of valid criteria are:

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

**Comparing null Values**

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.

## 1.4. SQL Commands

There are 4 basic commands for manipulating data in SQL, corresponding to the CRUD create, read, update, and delete operations: INSERT, SELECT, UPDATE, and DELETE. In addition, procedures can be executed using the EXECUTE command or through a *procedural relational command*.

### 1.4.1. SELECT Command

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

A SELECT command has a number of clauses:

- *SELECT ...*
- *[FROM ...]*
- *[WHERE ...]*
- *[GROUP BY ...]*
- *[HAVING ...]*
- *[ORDER BY ...]*
- *[LIMIT [offset,] limit]*
- *[OPTION ...]*

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.

- 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 can be used to understand many questions about SQL. 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.

### 1.4.2. 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

### 1.4.3. UPDATE Command

The UPDATE command is used to modify records in a table. The operation may 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]

### 1.4.4. DELETE Command

The DELETE command is used to remove records from a table. The operation may 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]

### 1.4.5. 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, the same as is returned from a SELECT. Note that EXEC or CALL can be used as a short form of this command.

Example Syntax

- EXECUTE proc()
- EXECUTE 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 are 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.
- 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.

### 1.4.6. Cache Hint

Non-update user commands may be preceded with a cache hint, e.g. /* cache */ select ..., to inform the engine that the results of command should be cached. The scope of the result, either session or global, will be deteremined automatically from the determinism level of the source queries and functions executed.

ResultSet caching must be enabled for this hint to have an effect.

### 1.4.7. Procedural Relational Command

Procedural relational commands use the syntax of a SELECT to emulate an EXEC. In a procedural relational command a procedure group names is used in a FROM clause in place of a table. That procedure will be executed in place of a 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*.

**Multiple Execution**

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

**Alternative Syntax**

None of issues listed in the syntax rules above exist if a *nested table reference* is used.

## 1.5. Temp Tables

Teiid supports creating temporary,or "temp", tables. Temp tables are dynamically created, but are treated as any other physical table.

Temp tables can be defined implicitly by referencing them in a SELECT INTO or in an INSERT statement or explicitly with a CREATE TABLE statement. Implicitly created temp tables must have a name that starts with '#'.

Creation syntax:

- CREATE LOCAL TEMPORARY TABLE<temporary table name> (<column name> <data type>,...)
- SELECT <column name>,...INTO <temporary table name> FROM <table name>
- INSERT INTO <temporary table name> ((<column name>,...)VALUES (<value>,...)

Drop syntax:

- DROP TABLE <temporary table name>

Limitations:

- With the CREATE TABLE syntax only basic table definition (column name and type information) is 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 sesssion or the block of a virtual procedure that creates it.
- Session level temp tables are not fail-over safe.
- temp tables are non-transactional.

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 *virtual procedures* for more on temp table usage.

## 1.6. SQL Clauses

This section describes the clauses that are used in the various *SQL commands* described in the previous section. Nearly all these features follow standard SQL syntax and functionality, so any SQL reference can be used for more information.

### 1.6.1. SELECT Clause

SQL queries start with the SELECT keyword and are often referred to as "SELECT statements". Teiid 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.

### 1.6.2. 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
- FROM table1 JOIN table2 MAKEDEP ON join-criteria
- FROM table1 JOIN table2 MAKENOTDEP ON join-criteria
- FROM table1 left outer join */* optional */* table2 ON join-criteria
- FROM *TEXTTABLE...*
- FROM *XMLTABLE...*

**DEP Hints**

MAKEDEP and MAKENOTDEP are hints used to control *dependent join* behavior. 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.

### 1.6.2.1. Nested Table Reference

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 preceeding FROM clause column references as long as INNER and LEFT OUTER joins are used. This is especially useful in cases where then 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
```

**Multiple Execution**

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

### 1.6.2.2. TEXTTABLE

The TEXTTABLE funciton processes character input to produce tabular ouptut. 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 correlated to preceeding FROM clause entries.

Usage:

```
TEXTTABLE(expression COLUMNS <COLUMN>, ... [DELIMITER char] [(QUOTE|ESCAPE)
 char] [HEADER [integer]] [SKIP integer]) AS name
```

```
COLUMN := name datatype [WIDTH integer]
```

Parameters

- expression - the text content to process, which should be convertable to CLOB.

- 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. HEADER may still be specified with SKP.

Syntax Rules:

- If width is specified for one column it must be specified for all columns.
- If width is specified, then fixed width parsing is used and ESCAPE, QUOTE, and HEADER should not be specified.
- The columns names must be not contain duplicates.

Examples

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

```
select * from texttable('col1,col2,col3\na,b,c' COLUMNS col2 string HEADER) x
```

- Use of fixed width, returns 1 row ['a', 'b', 'c']:

```
select * from texttable('abc' COLUMNS col1 string width 1, col2 string width 1, col3 string width
 1) 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
```

### 1.6.2.3. XMLTABLE

The XMLTABLE funciton uses XQuery to produce tabular ouptut. The XMLTABLE function is implicitly a nested table and may be correlated to preceeding FROM clause entries. 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 [47]*.

See XMLQUERY for the definition of *PASSING [49]*.

See also *XMLQUERY*

Parameters

- The optional XMLNAMESPACES clause specifies the namepaces for use in the XQuery and COLUMN path expressions.
- The xquery-expression should 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. 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. A FOR ORDINALITY column is typed as integer and will return the 1-based item number as its value.

Syntax Rules:

- Only 1 FOR ORDINALITY column may be specified.
- The columns names must be not contain duplicates.

Examples

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

```
select * from xmltable('/a' PASSING {x '<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
```

## 1.6.3. WHERE Clause

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

The general form of the WHERE is:

- WHERE *criteria*

## 1.6.4. 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.

The general form of the GROUP BY is:

- GROUP BY expression (,expression)*

Syntax Rules:

- Column references in the group by clause must by to unaliased output columns.
- Expressions used in the group by must appear in the select clause.
- Column references and expessions 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.

## 1.6.5. 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.

## 1.6.6. 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], ...
```

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 Teiid optimizer.

**Warning**

The use of positional ordering is no longer supported by the ANSI SQL standard and is a deprecated feature in Teiid. It is preferable to use alias names in the order by clause.

### 1.6.7. 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.

Usage:

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

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)

### 1.6.8. 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.

When the into clause is specified with a SELECT, the results of the query are inserted into the specified table. This is often used to insert records into a temporary table. The INTO clause immediately precedes the FROM clause.

Usage:

```
INTO table FROM ...
```

Syntax Rules:

- The INTO clause is logically applied last in processing, after the ORDER BY and LIMIT clauses.
- Teiid's support for SELECT INTO is similar to MS SQL Server. The target of the INTO clause is a table where the result of the rest select command will be inserted. SELECT INTO should not be used UNION query.

### 1.6.9. OPTION Clause

The OPTION keyword denotes options the user can pass in with the command. These options are Teiid-specific and not covered by any SQL specification.

Usage:

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

Supported options:

- MAKEDEP table [(,table)*] - specifies source tables that should 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

Previous versions of Teiid accepted the PLANONLY, DEBUG, and SHOWPLAN option arguments. These are no longer accepted in the OPTION clause. Please see the *other commands* chapter for using these options.

## 1.7. Set Operations

Teiid supports the UNION, UNION ALL, INTERSECT, EXCEPT set operation as a way of combining the results of commands.

Usage:

```
command (UNION|INTERSECT|EXCEPT) [ALL] command [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.

## 1.8. Subqueries

A subquery is a 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 from the outer query.
- Uncorrelated subquery - a subquery that contains no references to the outer sub-query.

Supported subquery locations:

- *Subqueries in the FROM clause*
- *Subqueries in the WHERE/HAVING Clauses*
- Subqueries may be used in any expression or CASE CRITERIA in the SELECT clasue.

## 1.8.1. 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.

**Example 1.1. Example Subquery in FROM Clause (Inline 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
```

## 1.8.2. Subqueries in the WHERE and HAVING clauses

Subqueries supported in the criteria of the outer query include subqueries in an IN clause, subqueries using the ANY/SOME or ALL predicate quantifier, and subqueries using the EXISTS predicate.

**Example 1.2. Example Subquery in WHERE Using EXISTS**

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

The following usages of subqueries must each select only one column, but can return any number of rows.

**Example 1.3. Example Subqueries in WHERE Clause**

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

# XML SELECT Command

## 2.1. Overview

Complex XML documents can be dynamically constructed by Teiid 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 Designer guide for more on creating document models.

XML documents may also created via XQuery with the *XMLQuery* function or with various other *SQL/XML* functions.

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

## 2.2. Query Structure

A valid XML SELECT Command against a document model is of the form *SELECT ... FROM ... [WHERE ...] [ORDER BY ...]* . The use of any other SELECT command 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.

### 2.2.1. FROM Clause

Specifies the document to generate. Document names resemble other virtual groups - "model"."document name".

Syntax Rules:

- The from may only contain one unary clause specifying the desired document.

### 2.2.2. 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, it's attribute, and all child content should be output.

### 2.2.3. 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.
- Criteria that applies to a mapping class is associated to that mapping class via the *context* function. The absence of a context function implies the criteria applies to the root context.
- 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.

**Sibling Root Mapping Classes**

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.

### 2.2.3.1. XML SELECT Command Specific Functions

XML SELECT Command functions are resemble scalar functions, but act as hints in the WHERE clause. These functions are only valid in an XML SELECT Command.

#### 2.2.3.1.1. Context Function

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

Select the context for the containing conjunct.

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.

#### 2.2.3.1.2. Rowlimit Function

```
ROWLIMIT(arg)
```

Limits the rows processed for the given context.

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.

#### 2.2.3.1.3. Rowlimitexception 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.

### 2.2.4. 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 or the order by items is the relative order they will be applied to their respective mapping classes.

## 2.3. 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.

**Document Correctness**

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 generated a document that is not valid with respect to the schema. See *document validation* on how 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 to applying criteria and order bys to mapping classes.

### 2.3.1. Document Validation

The execution property XMLValidation should be set to 'true' to indicate that generated documents should be checked for correctness. Correctness checking will not prevent invalid documents from being generated, since correctness is checked after generation and not during.

# Datatypes

## 3.1. Supported Types

Teiid supports a core set of runtime types. Runtime types can be different than 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 3.1. Teiid 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 literal, e.g. varchar(100). | java.lang.String | VARCHAR | VARCHAR |
| 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 | numeric, integral type, signed 32-bit | java.lang.Integer | INTEGER | INTEGER |
| long or bigint | numeric, integral type, signed 64-bit | java.lang.Long | BIGINT | NUMERIC |
| biginteger | numeric, integral type, arbitrary precision of up to 1000 digits | java.lang.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.String | DOUBLE | DOUBLE |
|  |  | java.math.BigDecimal | NUMERIC | NUMERIC |

| Type | Description | Java Runtime Class | JDBC Type | ODBC Type |
|---|---|---|---|---|
| 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). | | | |
| 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 [a] | BLOB | VARCHAR |
| clob | character large object, representing a stream of characters | java.sql.Clob [b] | CLOB | VARCHAR |
| xml | XML document | java.sql.SQLXML [c] | JAVA_OBJECT | VARCHAR |

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

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

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

## 3.2. 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 datatype 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 Teiid 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 convertable.
- **Warning**

  The Teiid 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 should be taken to not assume the string form in criteria or other places where a variation may cause different results.

### Table 3.2. 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 |

| Source Type | Valid Implicit Target Types | Valid Explicit Target Types |
|---|---|---|
| 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 also *XMLPARSE*

[b]xml to string is equivalent to XMLSERIALIZE(exp AS STRING) - see also *XMLSERIALIZE*

## 3.3. Special Conversion Cases

### 3.3.1. Conversion of String Literals

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

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

Here if the created_by column has the datatype of date, Teiid automatically converts the string literal to a date datatype as well.

### 3.3.2. Converting to Boolean

Teiid can automatically convert literal strings and numeric type values to Boolean values as follows:

| Type | Literal Value | Boolean Value |
|---|---|---|
| String | 'false' | false |
| | 'unknown' | null |

| Type | Literal Value | Boolean Value |
|---|---|---|
| | other | true |
| Numeric | 0 | false |
| | other | true |

### 3.3.3. Date/Time/Timestamp Type Conversions

Teiid can implicitly convert properly formatted literal strings to their associated date-related datatypes 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` .

## 3.4. Escaped Literal Syntax

Rather than relying on implicit conversion, datatype 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 3.3. Escaped Literal Syntax**

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

# Scalar Functions

Teiid provides an extensive set of built-in scalar functions. See also *SQL Support* and *Datatypes* . In addition, Teiid provides the capability for user defined functions or UDFs. See the Developers Guide for adding UDFs. Once added UDFs may be called just like any other function.

## 4.1. Numeric Functions

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

| Function | Definition | Datatype Constraint |
|---|---|---|
| + - * / | Standard numeric operators | x in {integer, long, float, double, biginteger, bigdecimal}, return type is same as x [a] |
| 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 |
| 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 |

| Function | Definition | Datatype Constraint |
|---|---|---|
| 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 | Datatype 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 |

### 4.1.1. Parsing Numeric Datatypes from Strings

Teiid offers a set of functions you can use to parse numbers from strings. For each string, you need to provide the formatting of the string. These functions use the convention established by the java.text.DecimalFormat class to define the formats you can use with these functions. You can learn more about how this class defines numeric string formats by visiting the Sun Java Web site at the following *URL for Sun Java* [http://java.sun.com/javase/6/docs/api/java/text/DecimalFormat.html].

For example, you could use these function calls, with the formatting string that adheres to the java.text.DecimalFormat convention, to parse strings and return the datatype you need:

| Input String | Function Call to Format String | Output Value | Output Datatype |
|---|---|---|---|
| '$25.30' | | 25.3 | double |

| Input String | Function Call to Format String | Output Value | Output Datatype |
|---|---|---|---|
| | parseDouble(cost, '$#,##0.00;($#,##0.00)') | | |
| '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 |

### 4.1.2. Formatting Numeric Datatypes as Strings

Teiid offers a set of functions you can use to convert numeric datatypes into strings. For each string, you need to provide the formatting. These functions use the convention established within the java.text.DecimalFormat class to define the formats you can use with these functions. You can learn more about how this class defines numeric string formats by visiting the Sun Java Web site at the following *URL for Sun Java* [http://java.sun.com/javase/6/docs/api/java/text/DecimalFormat.html] .

For example, you could use these function calls, with the formatting string that adheres to the java.text.DecimalFormat convention, to format the numeric datatypes into strings:

| Input Value | Input Datatype | 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' |

## 4.2. 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 1-based. The 0 index is considered to be before the start of the string.

| 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 as input will return null. [a] | return type is integer |
| CHR(x) CHAR(x) | Provide the character for ASCII value x [a] | 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} |
| 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 {string}, return string |
| 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 white space | x in {string}, return string |

| Function | Definition | Datatype Constraint |
|---|---|---|
| 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 ''.<br><br>Names are optional for column reference expressions.<br><br>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 {string}, 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 white space | x is string, return string |
| SUBSTRING(x, y) | Get substring from x, from position y to the end of x | y in {integer} |
| SUBSTRING(x, y, 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 builtin Java Charset names are valid values for the encoding.[b] | 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.[b] | 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 | x in {string} |

| Function | Definition | Datatype Constraint |
|---|---|---|
| | character in z at the same position | |
| UCASE(x) | Uppercase of x | x in {string} |

[a]Non-ASCII range characters or integers used in these functions may produce different results or exceptions depending on where the function is evalutated (Teiid vs. source). Teiid's uses Java default int to char and char to int conversions, which operates over UTF16 values.

[b]See the *Charset JavaDoc* [http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html] for more on supported Charset names. For charsets, unmappable chars will be replaced with the charset default character. binary formats, such as BASE64, will error in their conversion to bytes is a unrecognizable character is encountered.

## 4.3. 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. You can learn more about how this class defines formats by visiting the Sun Java Web site at the following *URL for Sun Java* [http://java.sun.com/javase/6/docs/api/java/text/SimpleDateFormat.html].

| 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 | 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) | x in {date, timestamp}, returns integer |
| DAYOFYEAR(x) | Return Julian day number | x in {date, timestamp}, returns integer |
| 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) |

| Function | Definition | Datatype Constraint |
|---|---|---|
| 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 | 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 have the following definition:<br><br>1. SQL_TSI_FRAC_SECOND - fractional seconds (billionths of a second)<br>2. SQL_TSI_SECOND - seconds<br>3. SQL_TSI_MINUTE - minutes<br>4. SQL_TSI_HOUR - hours<br>5. SQL_TSI_DAY - days<br>6. SQL_TSI_WEEK - weeks<br>7. SQL_TSI_MONTH - months<br>8. SQL_TSI_QUARTER - quarters (3 months)<br>9. SQL_TSI_YEAR - years | 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} |
| TIMESTAMPDIFF(interval, startTime, endTime) | Calculate the approximate number of whole intervals in (endTime - startTime) using a specific interval type (as defined by the constants in TIMESTAMPADD). If (endTime > startTime), a positive number will be returned. If (endTime < startTime), a negative number will be returned. Calculations | Interval in {string}; startTime, endTime in {timestamp}, returns a long. |

| Function | Definition | Datatype Constraint |
|---|---|---|
| | are approximate and may be less accurate over longer time spans. | |
| WEEK(x) | Return week in year | x in {date, timestamp}, returns integer |
| YEAR(x) | Return four-digit year | x in {date, timestamp}, returns integer |

### 4.3.1. Parsing Date Datatypes from Strings

Teiid does not implicitly convert strings that contain dates presented in different formats, such as ‘19970101’ and ‘31/1/1996’ to date-related datatypes. You can, however, use the parseDate, parseTime, and parseTimestamp functions, described in the next section, to explicitly convert strings with a different format to the appropriate datatype. These functions use the convention established within the java.text.SimpleDateFormat class to define the formats you can use with these functions. You can learn more about how this class defines date and time string formats by visiting the *Sun Java Web site* [http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html] .

For example, you could use these function calls, with the formatting string that adheres to the java.text.SimpleDateFormat convention, to parse strings and return the datatype you need:

| String | Function Call To Parse String |
|---|---|
| '1997010' | 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') |

### 4.3.2. 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

## 4.4. Type Conversion Functions

Within your queries, you can convert between datatypes using the CONVERT or CAST keyword. See also *Data Type Conversions* .

| Function | Definition |
|---|---|
| CONVERT(x, type) | Convert x to type, where type is a Teiid Base Type |
| CAST(x AS type) | Convert x to type, where type is a Teiid Base Type |

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

## 4.5. Choice Functions

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

| Function | Definition | Datatype 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 compatable comparable types |

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

## 4.6. Decode Functions

Decode functions allow you to have the Teiid Server 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.

| Function | Definition | Datatype Constraint |
|---|---|---|
| DECODESTRING(x, y) | Decode column x using string of value pairs y and return the decoded column as a string | all string |
| DECODESTRING(x, y, z) | Decode column x using string of value pairs y with delimiter z and return the decoded column as a string | all string |

| Function | Definition | Datatype Constraint |
|---|---|---|
| DECODEINTEGER(x, y) | Decode column x using string of value pairs y and return the decoded column as an integer | all string parameters, return integer |
| DECODEINTEGER(x,y,z) | Decode column x using string of value pairs y with delimiter z and return the decoded column as an integer | 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 the Teiid System 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, the Teiid Server inserts a null into the result set.

When you use these DECODE functions, you can provide as many input/output value pairs if you want within the string. By default, the Teiid System 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 the Teiid Server found in that column.

## 4.7. Lookup Function

The Lookup function allows you to cache a table’s data in memory and access it through a scalar function. This caching accelerates response time to queries that use the lookup tables, known in business terminology as lookup tables or code tables.

A StatePostalCodes table used to translate postal codes to complete state names might represent an example of this type of lookup table. One column, PostalCode, represents a key column. Other tables refer to this two-letter code. A second column, StateDisplayName, would represent the complete name of the state. Hence, a query to this lookup table would typically provide the PostalCode and expect the StateDisplayName in response.

When you call this function for any combination of codeTable, returnColumn, and keyColumn for the first time, the Teiid System caches the result. The Teiid System uses this cached map for all queries, in all sessions, that later access this lookup table. The codeTable requires use of the fully-qualified name, and the returnColumn and keyColumn parameters should use shortened column names.

Because the Teiid System caches and indexes this information in memory, this function provides quick access after the Teiid System initially caches the lookup table. The Teiid System unloads these cached lookup tables when you stop and restart the Teiid System. Thus, you should not use this function for data that is subject to updates. Instead, you can use it against static data that does not change over time.

**Note**

- The keyColumn is expected to contain unique key values. If the column contains duplicate values, an exception will be thrown.
- Cached lookup tables might consume significant memory. You can limit the number and maximum size of these code tables by setting configuration properties.

| Function | Definition | Datatype Constraint |
|---|---|---|
| LOOKUP(codeTable, returnColumn, keyColumn, keyValue) | In the lookup table codeTable, find the row where keyColumn has the value keyValue and return the associated returnColumn | codeTable must be a fully-qualified string literal containing metadata identifiers, keyValue datatype must match datatype of the keyColumn, return datatype matches that of returnColumn. returnColumn and keyColumn parameters should use their shortened names. |

## 4.8. System Functions

System functions provide access to information in the Teiid system from within a query.

| Function | Definition | Datatype Constraint |
|---|---|---|
| `COMMANDPAYLOAD()` | Retrieve the string form of the command payload or null if no command payload was specified. The command payload is set by a method on the Teiid JDBC API extensions on a per-query basis. | Returns a string |
| `COMMANDPAYLOAD(key)` | Cast the command payload object to a java.util.Properties object and look up the specified key in the object | key in {string}, return is string |
| `ENV(key)` | Retrieve an environment property. The only key currently allowed is ‘sessionid’, although this will expand in the future. | key in {string}, return is string |

| Function | Definition | Datatype Constraint |
|---|---|---|
| `USER()` | Retrieve the name of the user executing the query | return is string |

## 4.9. XML Functions

XML functions provide functionality for working with XML data.

### 4.9.1. XMLCOMMENT

Returns an xml comment.

```
XMLCOMMENT(comment)
```

Comment is a string. Return value is xml.

### 4.9.2. XMLCONCAT

Returns an XML with the concatination of the given xml types.

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

Content is xml. Return value is xml.

If a value is null, it will be ignored. If all values are null, null is returned.

### 4.9.3. XMLELEMENT

Returns an XML element with the given name and content.

```
XMLELEMENT([NAME] name [, <NSP>] [, <ATTR>][, content]*)
```

```
ATTR:=XMLATTRIBUTES(exp [AS name] [, exp [AS name]]*)
```

```
NSP:=XMLNAMESPACES((uri AS prefix | DEFAULT uri | NO DEFAULT))+
```

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 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 a 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.

Name, 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.

*Example*: with an xml_value of <doc/>,

```
xmlelement('elem', 1, '<2/>', xml_value)
```

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

### 4.9.4. XMLFOREST

Returns an concatination of XML elements for each content item.

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

See XMLELEMENT for the definition of NSP - *XMLNAMESPACES [47]*.

Name is an identifier. Content can be any type. Return value is xml.

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.

### 4.9.5. XMLPARSE

Returns an XML type representation of the string value expression.

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

expr in {string, clob, blob}. Return value is xml.

If DOCIMENT is specfied 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.

### 4.9.6. XMLPI

Returns an xml processing instruction.

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

Name is an identifier. Content is a string. Return value is xml.

### 4.9.7. XMLQUERY

Returns the XML result from evaluating the given xquery.

```
XMLQUERY([<NSP>] xquery [<PASSING>] [(NULL|EMPTY) ON EMPTY]]
```

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

See XMLELEMENT for the definition of NSP - *XMLNAMESPACES [47]*.

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 evaluted 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.

XMLQUERY is part of the SQL/XML 2006 specification.

See also *XMLTABLE*

**Note**

A technique known as document projection is used to reduce the memory footprint of the context item document. Only the parts of the document needed by the xquery will be loaded into memory.

## 4.9.8. XMLSERIALIZE

Returns a character type representation of the xml expression.

```
XMLSERIALIZE([(DOCUMENT|CONTENT)] xml [AS datatype])
```

Return value mathces datatype.

Only a character type (string, varchar, clob) may be specified as the datatype. CONTENT is the default. If DOCUMENT is specified and the xml is not a valid document or fragment, then an exception is raised.

## 4.9.9. XSLTRANSFORM

Applies an XSL stylesheet to the given document.

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

Doc, xsl in {string, clob, xml}. Return value is a clob.

If either argument is null, the result is null.

### 4.9.10. XPATHVALUE

Applies the XPATH expression to the document and returns a string value for the first matching result.

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

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

An attempt is made to provide a meaningful result for non-text nodes.

## 4.10. Security Functions

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

### 4.10.1. HASROLE

Whether the current caller has the role roleName.

```
hasRole([roleType,] 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'

## 4.11. Nondeterministic Function Handling

Teiid 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. Session Deterministic - the function will return the same result for the given inputs under the same user session. This category includes the hasRole, env, and user functions. 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.

3. 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 evaulation will occur prior to pushdown - however multiple occurances of the same command deterministic time function are not guarenteed to evaluate to the same value.

4. 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 it's execution context (for example if the function is used in the select clause).

# Procedures

## 5.1. Procedure Language

Teiid 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* .

### 5.1.1. Command Statement

A command statement executes a *SQL command* , such as SELECT, INSERT, UPDATE, DELETE, or EXECUTE, against one or more data sources.

**Example 5.1. Example Command Statements**

```
SELECT * FROM MySchema.MyTable WHERE ColA > 100;
INSERT INTO MySchema.MyTable (ColA,ColB) VALUES (50, 'hi');
```

### 5.1.2. Dynamic SQL Command

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 STRING <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 "UVAR.". 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.

## Example 5.2. 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 STRING 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 5.3. 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 STRING ‘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;
...
```

Known Limitations and Work-Arounds

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

**Example 5.4. Example Assignment**

```
EXECUTE STRING <expression> AS x string INTO #temp;
DECLARE string VARIABLES.RESULT = SEELCT 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 5.5. 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 5.6. 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 5.7. Example with AS and INTO clauses**

```
/* This name does not need to match the expected update command symbol "count". */
EXECUTE STRING <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 the Designer.
- When using the "AS" clause only the type information will be available to the Designer. ResultSet columns generated from the "AS" clause then will have a default set of properties for length, precision, etc.

## 5.1.3. 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.

### 5.1.4. Assignment Statement

An assignment statement assigns a value to a variable by either evaluating an expression or executing a SELECT command that returns a column value from a single row.

Usage:

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

Example Syntax

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

### 5.1.5. If Statement

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

#### Example 5.8. 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 presense of a NULL value.

### 5.1.6. Loop Statement

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

Usage:

```
LOOP ON <select statement> AS <cursorname>
BEGIN
  ...
END
```

### 5.1.7. While Statement

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

Usage:

```
WHILE <criteria>
BEGIN
  ...
END
```

### 5.1.8. 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.

### 5.1.9. 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.

### 5.1.10. 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.

**Example 5.9. Example Error Statement**

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

## 5.2. Virtual Procedures

Virtual procedures are defined using the Teiid 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.

### 5.2.1. Virtual Procedure Definition

Usage:

```
CREATE VIRTUAL PROCEDURE
BEGIN
  ...
END
```

The CREATE VIRTUAL PROCEDURE line demarcates the beginning of the procedure. The BEGIN and END keywords are used to denote block boundaries. Within the body of the procedure, any valid *statement* may be used.

The last command statement executed in the procedure will be return as the result. The output of that statement must match the expected result set and parameters of the procedure.

### 5.2.2. Procedure Input Parameters

Virtual procedures can take zero or more input parameters. 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 an input to a virtual procedure by using the fully-qualified name of the param (or less if unambiguous). For example, MySchema.MyProc.Param1.

**Example 5.10. Example of Referencing an Input Parameter for 'GetBalance' Procedure**

```
CREATE VIRTUAL PROCEDURE
BEGIN
```

```
    SELECT Balance FROM MySchema.Accts WHERE MySchema.Accts.AccountID =
 MySchema.GetBalance.AcctID;
END
```

### 5.2.3. Example Virtual Procedures

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

**Example 5.11. 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 5.12. 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
```

### 5.2.4. Executing Virtual Procedures

You execute procedures using the SQL *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 just like any SELECT, so you can use this in many places you can use a SELECT. However, within a virtual procedure itself you cannot always use an EXEC directly. Instead, you use the following syntax:

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

The following are some examples of how you can use the results of a virtual procedure call within a virtual procedure definition:

- LOOP instruction - you can walk through the results and do work on a row-by-row basis
- Assignment instruction - you can run a command and set the first column / first row value returned to a variable
- `SELECT * INTO #temp FROM (EXEC ...) AS x` - you can select the results from a virtual procedure into a temp table, which you can then query against as if it were a physical table.

## 5.3. 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. Teiid 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.

## 5.3.1. Update Procedure Definition

Usage:

```
CREATE PROCEDURE
BEGIN
  ...
END
```

The CREATE VIRTUAL PROCEDURE line demarcates the beginning of the procedure. The BEGIN and END keywords are used to denote block boundaries. Within the body of the procedure, any valid *statement* may be used.

## 5.3.2. Special Variables

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

### 5.3.2.1. INPUT Variables

Every attribute in the view whose UPDATE and INSERT transformations you are defining has an equivalent variable named INPUTS.<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 null. In an INSERT procedure, the default value of these variables is the default value of the virtual table attributes, based on their defined types. See *CHANGING Variables* for distinguishing defaults from passed values.

**Warning**

In prior release of Teiid INPUT was also accepted as the quailifer for an input variable. As of Teidd 7, INPUT is a reserved word, so INPUTS is the preferred qualifier.

### 5.3.2.2. CHANGING Variables

Similar to INPUT Variables, 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.

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 |

#### 5.3.2.3. ROWS_UPDATED Variable

Teiid returns the value of the VARIABLES.ROWS_UPDATED variable as a response to an update command executed against the view. Your procedure must set the value that returns when an application executes an update command against the view, which triggers invocation of the update procedure. For example, if an UPDATE command is issued that affects 5 records, the ROWS_UPDATED should be set appropriately so that the user will receive '5' for the count of records affected.

### 5.3.3. Update Procedure Command Criteria

You can use a number of special SQL clauses when defining UPDATE or DELETE procedures. These make it easier to do variable substitutions in WHERE clauses or to check on the change state of variables without using a lot of conditional logic.

#### 5.3.3.1. HAS CRITERIA

You can use the HAS CRITERIA clause to check whether the user’s command has a particular kind of criteria on a particular set of attributes. This clause evaluates to either true or false. You can use it anywhere you can use a criteria within a procedure.

Usage:

```
HAS [criteria operator] CRITERIA [ON (column list)]
```

Syntax Rules

- The criteria operator, can be one of =, <, >, <=, >=, <>, LIKE, IS NULL, or IN.
- If the ON clause is present, HAS CRITERIA will return true only if criteria was present on all of the specified columns.
- The columns in a HAS CRITERIA ON clause always refer to view columns.

Some samples of the HAS CRITERIA clause:

| SQL | Result |
|---|---|
| `HAS CRITERIA` | Checks simply whether there was any criteria at all. |
| `HAS CRITERIA ON (column1, column2)` | Checks whether the criteria uses column1 and column2. |
| `HAS = CRITERIA ON (column1)` | Checks whether the criteria has a comparison criteria with = operator. |
| `HAS LIKE CRITERIA` | Checks whether the criteria has a match criteria using LIKE. |

The HAS CRITERIA predicate is most commonly used in an IF clause, to determine if the user issued a particular form of command and to respond appropriately.

### 5.3.3.2. TRANSLATE CRITERIA

You can use the TRANSLATE CRITERIA clause to convert the criteria from the user application's SQL command into the form required to interact with the target source or view tables. The TRANSLATE CRITERIA statement uses the SELECT transformation to infer the column mapping. This clause evaluates to a translated criteria that is evaluated in the context of a command.

Usage:

```
TRANSLATE [criteria operator] CRITERIA [ON (column list)] [WITH (mapping
 list)]
```

Syntax Rules

- The criteria operator, can be one of =, <, >, <=, >=, <>, LIKE, IS NULL, or IN.
- If the ON clause is present, TRANSLATE CRITERIA will only form criteria using the specified columns.
- The columns in a TRANSLATE CRITERIA ON clause always refer to view columns.

You can use these mappings either to replace the default mappings generated from the SELECT transformation or to specify a reverse expression when a virtual column is defined by an expression.

Some samples of the HAS TRANSLATE clause:

| SQL | Result |
|---|---|
| `TRANSLATE CRITERIA` | Translates any user criteria using the default mappings. |
| TRANSLATE CRITERIA WITH (column1 = 'A', column2 = INPUTS.column2 + 2) | Translates any criteria with some additional mappings: column1 is always mapped to 'A' |

| SQL | Result |
|---|---|
| | and column2 is mapped to the incoming column2 value + 2. |
| `TRANSLATE = CRITERIA ON (column1)` | Translates only criteria that have = comparison operator and involve column1. |

The TRANSLATE CRITERIA, ON clause always refers to view columns. The WITH clause always has items with form <elem> = <expression>, where the <elem> is a view column and the <expression> specifies what that view column should be replaced with when TRANSLATE CRITERIA translates the view criteria (from UPDATE or DELETE) into a physical criteria in the command. By default, a mapping is created based on the SELECT clause of the SELECT transformation (view column gets mapped to expression in SELECT clause at same position).

### 5.3.4. 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, as 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.

# Transaction Support

Teiid utilizes XA transactions for participating in global transactions and for demarcating its local and command scoped transactions. *JBoss Transactions* [http://www.jboss.org/jbosstm/] is used by Teiid as its transaction manager. See *this documentation* [http://www.jboss.org/jbosstm/docs/index.html] for the advanced features provided by JBoss Transactions.

### Table 6.1. Teiid 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 | Teiid participates in a global transaction as an XA Resource. |

The default transaction isolation level for Teiid is READ_COMMITTED.

## 6.1. 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 Teiid based upon command type, the transaction isolation level, and available metadata. A wrapping transaction is not needed if:

- If a user command is fully pushed to the source.

- If the user command is a SELECT (including XML) and the transaction isolation is not REPEATABLE_READ nor SERIALIABLE.
- If the user command is a stored procedure and the transaction isolation is not REPEATABLE_READ nor SERIALIABLE 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.

## 6.2. 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 of 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. |

## 6.3. JDBC and Transactions

### 6.3.1. JDBC API Functionality

The transaction scopes above map to these 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 Teiid is consumed through its XADataSource, `org.teiid.jdbc.TeiidDataSource` . JEE containers or data access APIs typically control XA transactions on behalf of application code.

### 6.3.2. 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 app 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 app 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.

## 6.4. Limitations and Workarounds

- The client setting of transaction isolation level is not propogated 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.
- Temporary tables are not transactional. For example, a global temporary table will retain all inserts performed during a local transaction that was rolled back.

# Data Roles

Data roles, also called entitlements, are sets of permissions that are defined per VDB that dictate data access (create, read, update, delete). The use of data roles is controlled system wide with the property in `<jboss-install>/server/<profile>/deploy/teiid/teiid-jboss-beans.xml` file in bean configuration section of `RuntimeEngineDeployer` with property `useEntitlements`.

Once data roles are enabled, the access permissions defined in a VDB will be enforced by the Teiid Server.

## 7.1. 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.

## 7.2. XML Definition

Data roles are defined inside the `vdb.xml` file (inside the .vdb Zip archive under META-INF/vdb.xml) if you used Designer. This example will show a sample "vdb.xml" file with few simple data rules.

For example, if a VDB defines a table "TableA" in schema "modelName" with columns (column1, column2) - note that the column types do not matter. And we wish to define three roles "RoleA", "RoleB", "RoleC" with following permissions:

1. RoleA has privileges to read, write access to TableA, but can not delete.
2. RoleB has no privileges that allow access to TableA
3. RoleC has privileges 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-policy name="RoleA">
        <description>Allow all, except Delete</description>

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

        <permission>
            <resource-name>modelName.TableA.colum1</resource-name>
            <allow-create />
            <allow-read />
            <allow-update />
        </permission>

        <permission>
            <resource-name>modelName.TableA.column2</resource-name>
            <allow-create />
            <allow-read />
            <allow-update />
        </permission>
        <mapped-role-name>role1</mapped-role-name>

    </data-policy>

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

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

        <permission>
            <resource-name>modelName.TableA.colum1</resource-name>
            <allow-read />
        </permission>

        <mapped-role-name>role2</mapped-role-name>
    </data-policy>
</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 Teiid uses deny by default, there is no explict data-policy entry needed for "RoleB". The "mapped-role-name" defines the "role" to whom these policies are applicable. Each data-policy must define a "role" to be enforced by the Teiid Server.

For assigning the roles to your users, in the JBoss AS, check out the instructions for the selected Login Module. Check "Admin Guide" for configuring Login Modules.

"vdb.xml" file is checked against the schema file `vdb-deployer.xsd`, check the documents sections of the Teiid kit to find a copy of the schema file.

**Note**

Currently there is no GUI tooling support in the Designer or any other management tool to create this data roles permissions xml, however this is in our roadmap for future releases to provide.

## 8.1. System Tables

### 8.1.1. VDB, Schema, and Properties

#### 8.1.1.1. 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 |

#### 8.1.1.2. 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 |
| Description | string | Description |
| PrimaryMetamodelURI | string | URI for the primary metamodel describing the model used for this schema |

#### 8.1.1.3. 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 |

### 8.1.2. Table Metadata

#### 8.1.2.1. 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 |
| 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 |
| Cardinality | integer | Approximate number of rows in the group |
| Description | string | Description |
| IsSystem | boolean | True if in system table |

### 8.1.2.2. 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 | Teiid runtime data type name |
| Scale | integer | Number of digits after the decimal point |
| 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 |

| Column Name | Type | Description |
|---|---|---|
| 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 numeric value |
| MaxRange | string | Maximum numeric value |
| 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 |
| Radix | integer | Radix for numeric values |
| GroupUpperName | string | Upper-case full group name |
| UpperName | string | Upper-case element name |
| UID | string | Element unique ID |
| Description | string | Description |

### 8.1.2.3. 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 |

| Column Name | Type | Description |
|---|---|---|
| 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 |

#### 8.1.2.4. 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 |
| Position | integer | Position in key |

### 8.1.3. Procedure Metadata

#### 8.1.3.1. 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 |
| ReturnsResults | boolean | Returns a result set |
| UID | string | Procedure UID |
| Description | string | Description |

#### 8.1.3.2. 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 | Teiid 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" |

### 8.1.4. Datatype Metadata

#### 8.1.4.1. SYS.DataTypes

This table supplies information on *datatypes* .

| Column Name | Type | Description |
|---|---|---|
| Name | string | Teiid 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" |

| Column Name | Type | Description |
|---|---|---|
| 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 |
| RuntimeType | string | Teiid runtime data type name |
| BaseType | string | Base type |
| Description | string | Description of type |

## 8.2. System Procedures

| Procedure | Parameters | ResultSet |
|---|---|---|
| getCharacterVDBResource | (string resourcePath) | A single column containing the resource as a clob. |
| getBinaryVDBResource | (sting resourcePath) | A single column containing the resource as a blob. |
| getVDBResourcePaths | () | A single column containing the resource paths as strings. |
| getXMLSchemas | (string document) | A single column containing the schemas as clobs. |

# Translators

## 9.1. Introduction to the Teiid Connector Architecture

The TCA (Teiid Connector Architecture) provides Teiid with a robust mechanism for integrating with external systems. The TCA defines a common client interface between Teiid and an external system that includes metadata as to what SQL constructs are supported for pushdown and the ability to import metadata from the external system.

>A Translator is the heart of the TCA and acts as the bridge logic between Teiid and an external system, which is most commonly accessed through a JCA resource adapter. See the Teiid Developers Guide for details on developing custom Translators and JCA resource adapters for use with Teiid.

**Note**

The TCA is not the same as the JCA, the JavaEE Connector Architecture, although the TCA is designed for use with JCA resource adapters.

**Note**

The import capabilities of Teiid Translators is currently only used in *dynamic VDBs* and not by the Teiid Designer.

## 9.2. Translators

A Translator is typically paired with a particular JCA resource adapter. In instances where pooling, environment dependent configuration management, advanced security handling, etc. are not needed, then a JCA resource adapter is not needed. The configuration of JCA ConnectionFactories for needed resource adapters is not part of this guide, please see the Teiid Admin Guide and the kit examples for configuring resource adapters for use in JBossAS.

Translators can have a number of configurable properties. These are broken down into execution properties, which determine aspects of how data is retrieved, and import settings, which determine what metadata is read for import.

The execution properties for a translator typically have reasonable defaults. For specific translator types, e.g. the Derby translator, base execution properties are already tuned to match the source. In most cases the user will not need to adjust their values.

### Table 9.1. Base Execution Properties - shared by all translators

| 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. | false |
| SupportsOuterJoins | Set to true to indicate that OUTER JOINs are supported. | false |
| SupportsFullOuterJoins | If outer joins are supported, true indicates that FULL OUTER JOINs are supported. | false |
| SupportsInnerJoins | Set to true to indicate that INNER JOINs are supported. | false |
| 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 |

**Note**

Only a subset of the metadata as to what SQL constructs the source supports can be set through execution properties. If more control is needed, please consult the Teiid Developers Guide.

There are no base importer settings.

## 9.2.1. File Translator

The file translator, known by the type name *file*, 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 formated data.

### Table 9.2. Execution Properties

| Name | Description | Default |
|---|---|---|
| Encoding | The encoding that should be used for CLOBs returned by the getTextFiles procedure | The system default encoding |

There are file importer settings, but it does provide metadata for dynamic vdbs.

#### 9.2.1.1. Usage

Retrieve all files as BLOBs with the given extension at the given path.

```
call getFiles('path/*.ext')
```

If the extension pattern is not specified and the path is a directory, then all files in the directory will be returned. If the path or filename doesn't exist, then no results will be returned.

Retrieve all files as CLOBs with the given extension at the given path.

```
call getTextFiles('path/*.ext')
```

Save the CLOB, BLOB, or XML file to given path

```
call saveFile('path', value)
```

See the database metadata for full descriptions of the getFiles, getTextFiles, and saveFile procedures.

### 9.2.2. JDBC Translator

The JDBC translator bridges between SQL semantic and data type difference between Teiid and a target RDBMS. Teiid has a range of specific translators that target the most popular open source and proprietary databases.

Type names

- *jdbc-ansi* - declares support for most SQL constructs supported by Teiid, except for row limit/ offset and EXCEPT/INTERCECT. Translates source SQL into ANSI compliant syntax. This translator should be used when another more specific type is not available.
- *jdbc-simple* - same as jdbc-ansi, except disables support for function, UNION, and aggregate pushdown.
- *db2* - for use with DB2 8 or later.
- *derby* - for use with Derby 10.1 or later. Setting the specific 10.x version improves pushdown support - see the DatabaseVersion execution property.
- *h2* - for use with H2 version 1.1 or later.

- *hsql* - for use with HSQLDB 1.7 or later.
- *informix* - for use with any version.
- *metamatrix* - for use with MetaMatrix 5.5.0 or later.
- *mysql*/*mysql5* - for use with MySQL version 4.x and 5 or later respectively.

> **Note**
>
> The MySQL Translators expect the database or session to be using ANSI mode. If the database is not using ANSI mode, an initialization query should be used on the pool to set ANSI mode:
>
> ```
> set SESSION sql_mode = 'ANSI'
> ```

- *oracle* - 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 <sequencesequence name>.[nextval|currentval]. You can use a sequence as the default value for insert columns by setting the column to autoincrement and the name in source to <element name>:SEQUENCE=<sequence name>.<sequence value>.
- *postgresql* - for use with 8.0 or later clients and 7.1 or later server. Setting the specific 8.x version improves pushdown support - see the DatabaseVersion execution property.
- *sybase* - for use with Sybase version 12.5 or later.
- *teiid* - for use with Teiid 6.0 or later.
- *teradata* - for use with Teradata V2R5.1 or later.

### Table 9.3. Execution Properties - shared by all JDBC Translators

| Name | Description | Default |
|---|---|---|
| DatabaseTimeZone | The time zone of the database. Used when fetchings 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 |
| TrimStrings | true to trim trailing whitespace from fixed length character strings. Note that Teiid only has a string, or varchar, type that treats trailing whitespace as meaningful. | false |

| Name | Description | Default |
|---|---|---|
| UseBindVariables | true to indicate that PreparedStatements should be used and that literal values in the source query should be replace with bind variables. If false only LOB values will trigger the use of PreparedStatements. | true |
| UseCommentsInSourceQuery | This will embed a /*comment*/ leading comment with session/request id in source SQL query for informational purposes | false |

### Table 9.4. Importer Properties - shared by all JDBC Translators

| Name | Description | Default |
|---|---|---|
| catalog | See DatabaseMetaData.getTables[1] | null |
| schemaPattern | See DatabaseMetaData.getTables[1] | null |
| tableNamePattern | See DatabaseMetaData.getTables[1] | null |
| procedurePatternName | See DatabaseMetaData.getProcedures[1] | null |
| tableTypes | Comma separated list - without spaces - of imported table types. See DatabaseMetaData.getTables[1] | null |
| useFullSchemaName | When false, directs the importer to drop the source catalog/schema from the Teiid object name, so that the Teiid fully qualified name will be in the form of <model name>.<table name> - Note: that this may lead to objects with duplicate names when importing from multiple schemas, which results in an exception | true |
| importKeys | true to import primary and foriegn keys | true |
| importIndexes | true to import index/unique key/cardinality information | true |
| importApproximateIndexes | true to import approximate index information. See DatabaseMetaData.getIndexInfo[1] | true |
| importProcedures | 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 | 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 | | true |

| Name | Description | Default |
|---|---|---|
| | false will override the default and direct Teiid to create source queries using unquoted identifiers. | |

[1]Full JavaDoc for *DatabaseMetaData* [http://java.sun.com/javase/6/docs/api/java/sql/DatabaseMetaData.html]

**Warning**

The default import settings will crawl all available metadata. This import process is time consuming and full metadata import is not needed in most situations. Most commonly you'll want to limit import by schemaPattern and tableTypes.

Example importer settings to only import tables and views from my-schema.

```
...
<property name="importer.tableTypes" value="TABLE,VIEW"/>
<property name="importer.schemaPattern" value="my-schema"/>
...
```

#### 9.2.2.1. Usage

Usage of a JDBC source is straight-forward. Using Teiid SQL, the source ma be queried as if the tables and procedures were local to the Teiid system.

### 9.2.3. LDAP Translator

The LDAP translator, known by the type name *ldap*, exposes an LDAP directory tree relationally with pusdown support for filtering via criteria. This is typically coupled with the LDAP resource adapter.

**Table 9.5. Execution Properties**

| Name | Description | Default |
|---|---|---|
| SearchDerfaultBaseDN | 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 |

### 9.2.4. 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 configurable behavior.

**Table 9.6. Execution Properties**

| Name | Description | Default |
|---|---|---|
| ThrowError | true to always throw an error | false |
| RowCount | Rows returned for non-update queries. | 1 |
| WaitTime | Wait randomly up to this number of milliseconds with each sourc query. | 0 |
| PollIntervalInMilli | if positive results will be "asynchronusly" returned - that is a DataNotAvailableException will be thrown initially and the engine will wait the poll interval before polling for the results. | -1 |

There are no import settings for the Loopback translator; it also does not provide metadata - it should be used as a testing stub.

### 9.2.5. Salesforce Translator

The Salesforce translator, known by the type name *salesforce* supports the SELECT, DELETE, INSERT and UPDATE operations against a Salesforce.com account. It is designed for use with the Teiid Salesforce resource adapter.

**Table 9.7. Execution Properties**

| Name | Description | Default |
|---|---|---|
| ModelAuditFeilds | Audit Model Fields | false |

The Salesforce translator can import metadata, but does not currently have import settings.

#### 9.2.5.1. Usage

##### 9.2.5.1.1. 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 Teiid Query Planner, the Salesforce connector supports nearly all of the SQL syntax supported by the Teiid.

The Salesforce Connector executes SQL commands by “pushing down” the command to Salesforce whenever possible, based on the supported capabilities. Teiid 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 down" to the data source, so it will be evaluated in Teiid, in order to ensure that the operation is performed.

In cases where certain SQL capabilities cannot be pushed down to Salesforce, Teiid will push down the capabilities that are supported, and fetch a set of data from Salesforce. Then, Teiid will evaluate the additional capabilities, creating a subset of the original data set. Finally, Teiid 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 Teiid 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's useful to be aware of unsupported capabilities, in order to avoid fetching large data sets from Salesforce and making you queries as performant as possible. See all *Supported Capabilities*.

#### 9.2.5.1.2. Selecting from Multi-Select Picklists

A multi-select picklist is a field type in Salesforce that can contain multiple values in a single field.

Query criteria operators for fields of this type in SOQL are limited to EQ, NE, includes and excludes. The full Salesforce documentation for selecting from mullti-select picklists can be found at the following link. *Querying Mulit-select Picklists* [http://www.salesforce.com/us/developer/docs/api/index_Left.htm#StartTopic=Content%2Fsforce_api_calls_soql_querying_multiselect_picklists.htm|SkinName=webh

Teiid SQL does not support the includes or excludes operators, but the Salesforce connector provides user defined function definitions for these operators that provided equivalent functionality for fields of type multi-select. The definition for the functions is:

```
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';
```

### 9.2.5.1.3. Selecting All Objects

The Salesforce connector supports the 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 perform the default behavior will call query.

```
  select * from Contact where isDeleted = false;
```

#### 9.2.5.1.4. Selecting Updated Objects

If the option is selected when importing metadata from Salesforce, a GetUpdated procedure is generated in the model with the following sturcture:

```
GetUpdated (ObjectName IN string,
 StartDate IN datetime,
 EndDate IN datetime,
 LatestDateCovered OUT datetime)
returns
 ID string
```

See the description of the *GetUpdated* [http://www.salesforce.com/us/developer/docs/api/Content/sforce_api_calls_getupdated.htm] operation in the Salesforce documentation for usage details.

#### 9.2.5.1.5. Selecting Deleted Objects

If the option is selected when importing metadata from Salesforce, a GetDeleted procedure is generated in the model with the following sturcture:

```
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* [http://www.salesforce.com/us/developer/docs/api/Content/sforce_api_calls_getdeleted.htm] operation in the Salesforce documentation for usage details.

#### 9.2.5.1.6. 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* [http://www.salesforce.com/us/developer/docs/api/index_Left.htm#StartTopic=Content/sforce_api_calls_soql_relationships.htm] operation in the SalesForce documentation for limitations.

### 9.2.5.1.7. Supported Capabilities

The following are the 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

## 9.2.6. Web Services Translator

The Web Services translator, known by the type name *ws*, exposes stored procedures for calling web services backed by a Teiid WS resource adapter. It will commonly be used with the *TEXTTABLE* or *XMLTABLE* table functions to use CSV or XML formated data.

**Table 9.8. Execution Properties**

| Name | Description | Default |
|---|---|---|
| DefaultBinding | The binding that should be used if one is not specified. Can be one of HTTP, SOAP11, or SOAP12 | SOAP12 |

| Name | Description | Default |
|---|---|---|
| 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 | 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. | null - unused |

There are ws importer settings, but it does provide metadata for dynamic vdbs.

**Note**

Setting the org.teiid.CONNECTOR.WS logging context to detail will show the request/response documents as part of the log.

### 9.2.6.1. Usage

The main 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's 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. e.g.

```
call invoke(binding='HTTP', action='GET')
```

The request XML should be a valid XML document or root element.

See the ws-weather example in the kit and database metadata for a full description of invoke.

## 9.3. Dynamic VDBs

Teiid integration is available via a "Dynamic VDB" without the need for Teiid Designer tooling. While this mode of operation does not yet allow for the creation of view layers, the underlying sources can still be queried as if they are a single source. See the kit's "teiid-example/dynamicvdb-*" for working examples.

To build a dynamic vdb, you'll need to create a "<some-name>-vdb.xml" file. The XML file captures information about the VDB, the sources it integrate, and preferences for importing metadata.

**Note**

VDB name pattern must adhere to "-vdb.xml" for the Teiid VDB deployer to recognize this file as a dynamic VDB.

my-vdb.xml: (The vdb-deployer.xml schema for this file is available in the schema folder under the docs with the Teiid distribution.)

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

  <property name="UseConnectorMetadata" value="..."/>

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

    <property name="..." value="..." />
    ...

    <source name="${source-name}" translator-name="${translator-name}"

    connection-jndi-name="${deployed-jndi-name}">
  </model>

  <!-- create translator instances that override default properties -->

  <translator name="${translator-name}" type="${translator-type}"/>

    <property name="..." value="..."/>
    ...
    </translator>
</vdb>
```

### 9.3.1. 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

- *UseConnectorMetadata* - Setting to use connector supplied metadata. Can be "true" or "cached". "true" will obtain metadata once for every launch of Teiid. "cached" will save a file containing the metadata into the <jboss-install>/server/<profile>/data/teiid directory

### 9.3.2. 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 should be unique among all Models in the VDB and should not contain the '.' character.
- *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.

#### Source Element

- *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 than the model name is only useful in multi-source scenarios.
- *translator-name* - The name or type of the Teiid 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 AS. 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.

## 9.3.3. Translator Element

### Attributes

- *name* - The name of the 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.

# Federated Planning

Teiid at its core 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, not on hand-coding joins, and other relational operations, between data sources.

## 10.1. Overview

When the query engine receives an incoming SQL query it performs the following operations:

1. Parsing - validate syntax and convert to internal form
2. Resolving - link all identifiers to metadata and functions to the function library
3. Validating - validate SQL semantics based on metadata references and type signatures
4. Rewriting - rewrite SQL to simplify expressions and criteria
5. Logical plan optimization - the rewritten canonical SQL is converted into a logical plan for in-depth optimization. The Teiid optimizer is predominantly rule-based. Based upon the query structure and hints a certain rule set will be applied. These rules may trigger in turn trigger the execution of more rules. Within several rules, Teiid also takes advantage of costing information. The logical plan optimization steps can be seen by using the *OPTION DEBUG* clause and are described in the *query planner* section.
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* .

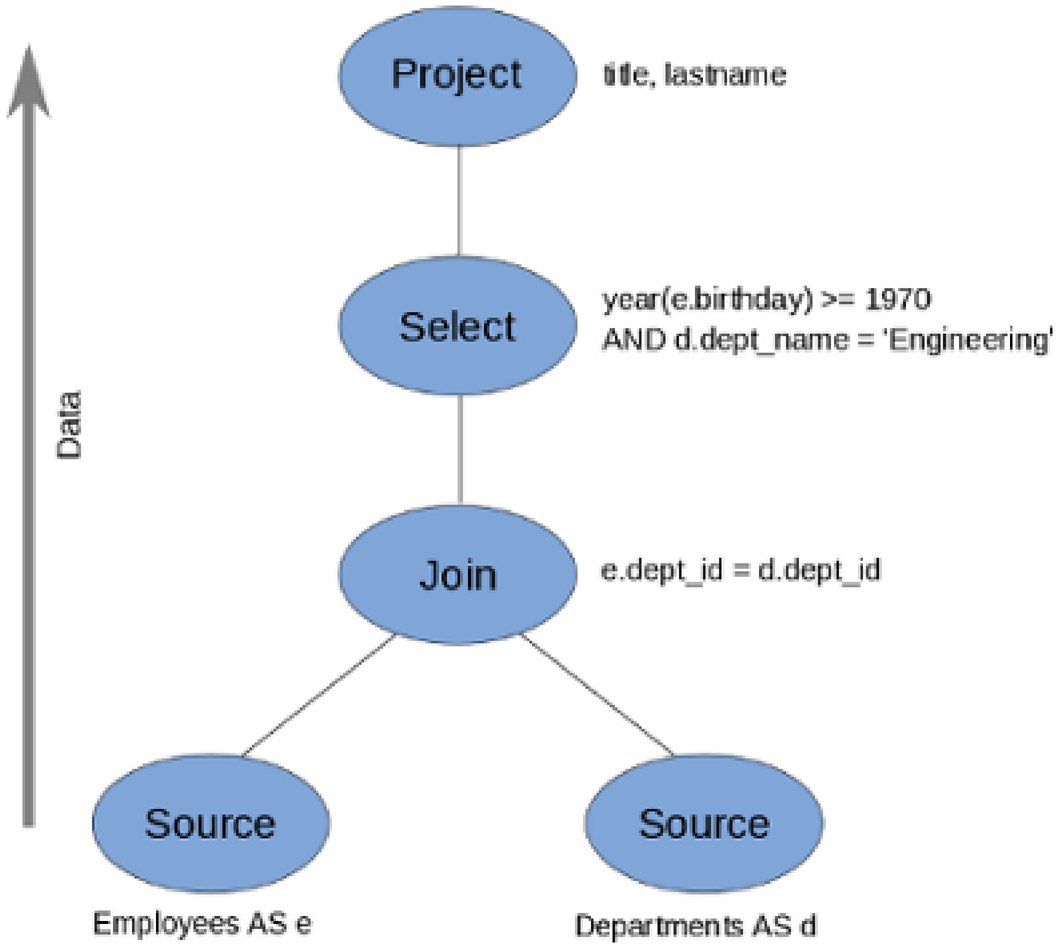
The logical query plan is a tree of operations used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute column values), *join* , *source* (retrieve data from a table), *sort* (ORDER BY), *duplicate removal* (SELECT DISTINCT), *group* (GROUP BY), and *union* (UNION).

For example, consider the following query that retrieves all engineering employees born since 1970.

### Example 10.1. 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:



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.

Of course, 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.

## 10.2. Federated Optimizations

### 10.2.1. 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 run-away 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.

### 10.2.2. Pushdown

In federated database systems pushdown refers to decomposing the user level 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 Teiid though the Connector API. Any work not performed at the source is then processed in Federate's relational engine.

Based upon capabilities, Teiid will manipulate the query plan to ensure that each source performs as much joining, filtering, grouping, etc. as possible. In may cases, such as with join ordering, planning is a combination of *standard relational techniques* and, cost based and heuristics for pushdown optimization.

Criteria and join push down are typically the most important aspects of the query to push down when performance is a concern. See *Query Plans* on how to read a plan to ensure that source queries are as efficient as possible.

### 10.2.3. 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 drastically 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.

Teiid supports the MAKEDEP and MAKENOTDEP hints. Theses are can be placed in either the *OPTION clause* or directly in the *FROM clause* . As long as all can be met, the MAKEDEP and MAKENOTDEP hints override any use of costing information.

**Tip**

The MAKEDEP hint should only be used if the proper query plan is not chosen by default. You should ensure that your costing information is representative of the actual source cardinality. An inappropriate MAKEDEP hint can force an inefficient join structure and may result in many source queries.

### 10.2.4. 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

### 10.2.5. Projection Minimization

Teiid 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.

### 10.2.6. 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.

### 10.2.7. Optional Join

The optional join hint indicates to the optimizer that a join clause should be omitted if none of its columns are used in either user criteria or output columns in the result. 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.

#### Example 10.2. Example Optional Join Hint

```
select a.column1, b.column2 from a inner join /* optional */ b on a.key = b.key
```

Suppose that the preceding example defined 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
```

**Tip**

When a join clause is omitted via the optional join hint, the relevant join 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 does not require a hint.

### 10.2.8. Standard Relational Techniques

Teiid 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.

## 10.3. Federated Failure Modes

### 10.3.1. Partial Results

Teiid provides the capability to obtain "partial results" in the event of data source unavailability. 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 info is not available.

If one or more data sources are unavailable to return results, then the result set obtained from the remaining available sources will be returned. In the case of joins, an unavailable data source essentially contributes zero tuples to the result set.

#### 10.3.1.1. Setting Partial Results Mode

Partial results mode is off by default but can be turned on by default for all queries in a Connection with either setPartialResultsMode("true") on a DataSource or partialResultsMode=true on a JDBC URL. In either case, partial results mode may be overridden with a set statement.

### Example 10.3. Example - Setting Partial Results Mode

```
Statement statement = ...obtain statement from Connection...
statement.execute("set partialResultsMode true");
```

### 10.3.1.2. Source Unavailability

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 in the result set.

**Warning**

Since Teiid supports multi-source cursoring, it is possible that the unavailability of a data source will not be determined until after the first batch of results have been returned to the client. This can happen in the case of unions, but not joins. In this situation, there will be no warnings in the result set when the client is processing the first batch of results. The client will be responsible for periodically checking the status of warnings in the results object as results are being processed, to see if a new warning has been added due to the detection of an unavailable source. [Note that client applications have no notion of ‘batches’, which are purely a server-side entity. Client apps deal only with records.]

For each source that is excluded from a query, a warning will be generated describing the source and the failure. These warnings can be obtained from the Statement.getWarnings() method. This method returns a SQLWarning object but in the case of "partial results" warnings, this will be an object of type org.teiid.jdbc.PartialResultsWarning. This class can be used to obtain a list of all the failed connectors by name and to obtain the specific exception thrown by each connector.

### Example 10.4. Example - Printing List of Failed Sources

```
statement.execute("set partialResultsMode true");
ResultSet results = statement.executeQuery("SELECT Name FROM Accounts");
SQLWarning warning = statement.getWarnings();
if(warning instanceof PartialResultsWarning) {
  PartialResultsWarning partialWarning = (PartialResultsWarning)warning;
  Collection failedConnectors = partialWarning.getFailedConnectors();
  Iterator iter = failedConnectors.iterator();
  while(iter.hasNext()) {
    String connectorName = (String) iter.next();
   SQLException connectorException = partialWarning.getConnectorException(connectorName);
    System.out.println(connectorName + ": " + ConnectorException.getMessage();
  }
}
```

## 10.4. 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.

### 10.4.1. Getting a Query Plan

You can get a query plan any time you execute a command. The SQL options available are as follows:

- SHOWPLAN [ON|DEBUG]- Returns the 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.

**Example 10.5. 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 of Teiid's tools.

### 10.4.2. 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 that got pushed to each source
- Join ordering
- Join algorithm used - merge or nested loop.
- 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.

## 10.4.3. 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 Select.]

- Project - Defines the columns returned from the node. This does not alter the number of records returned. [When there is a subquery in the Select clause, this node is called Dependent Project.]

- 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.]

- Join - Defines the join type, join criteria, and join strategy (merge or nested loop).

- Union - There are no properties for this node, it just passes rows through from it's children
- Sort - Defines the columns to sort on, the sort direction for each column, and whether to remove duplicates or not.

- Dup Removal - Same properties as for Sort, but the removeDups property is set to true

- Group - 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.
- Limit - Returns a specified number of rows, then stops processing. Also processes an offset if present.

#### 10.4.3.1. 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.

| Statistic | Description | Units |
|---|---|---|
| Node Output Rows | Number of records output from the node | count |
| Node 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.

| Cost Estimates | Description | Units |
|---|---|---|
| Estimated Node Cardinality | Estimated number of records that will be output from the node; -1 if unknown | count |

## 10.5. Query Planner

For each sub-command in the user command an appropriate kind of sub-planner is used (relational, XML, procedure, etc).

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

## 10.5.1. Relational Planner

The GenerateCanonical class generates the initial (or “canonical” plan). This plan is based on the typical logical order that a SQL query gets executed. A SQL select query has the following possible clauses (all but SELECT are optional): SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. These clauses are logically executed in the following order:

1. FROM (read and join all data from tables)

2. WHERE (filter rows)

3. GROUP BY (group rows into collapsed rows)

4. HAVING (filter grouped rows)

5. SELECT (evaluate expressions and return only requested columns)

6. INTO

7. ORDER BY (sort rows)

8. LIMIT (limit result set to a certain range of results)

These clause translate into the following types of planning nodes:

- FROM: Source node for each from clause item, Join node (if >1 table)
- WHERE: Select node
- GROUP BY: Group node
- GROUP BY: Group node
- SELECT: Project node and DupRemoval node (for SELECT DISTINCT)
- INTO: Project node with a SOURCE Node
- INTO: Project node with a SOURCE Node
- LIMIT: Limit node
- UNION, EXCEPT, INTERSECT: SetOp Node

There is also a Null Node that can be created as the result of rewrite or planning optimizations. It represents a node that produces no rows

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 its transformations. For example, if there are no view layers, then RuleMergeVirtual, 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 *plan nodes* in contrast typical have fixed properties, and only allow for binary operations - due to algorithmic limitations.

Below are some of the rules included in the planner:

- RuleRemoveSorts - removes sort nodes that do not have an effect on the result.  This most common when a view has an non-limited ORDER BY.
- RulePlaceAccess - insert an Access node above every physical Source node.  The source node represents a table typically.  An access node represents the point at which everything below the access node gets pushed to the source.  Later rules focus on either pushing stuff under the access or pulling the access node up the tree to move more work down to the data sources.  This rule is also responsible for placing .
- RulePushSelectCriteria - pushes select criteria down through unions, joins, and views into the source below the access node.  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 PushSelectCriteria.  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)
```

  Note: 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. For example:

```
     SELECT (B.y = 3)
      |
     JOIN - Right Outer Join on (A.x = B.x)
    /     \
  SRC (A)   SRC (B)
```

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. For example:

```
     SELECT (B.y = 3)
      |
     JOIN - Left Outer Join on (A.x = B.x)
    /     \
  SRC (A)   SRC (B)
```

Can become (available only after 5.0.2):

```
     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.

On the other hand, criteria that are dependent upon the presence of null values CANNOT be moved.  For example:

```
      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.  This will be true regardless of which version of Teiid is used.

- RulePushNonJoinCriteria – this rule will push criteria out of an on clause if it is not necessary for the correctness of the join.
- RuleRaiseNull – this rule will raise null nodes to their highest possible point.  Raising a null node removes the need to consider any part of the old plan that was below the null node.
- RuleMergeVirtual - merges view layers together.  View layers are connected by nesting canonical plans under source leaf nodes of the parent plan.  Each canonical plan is also sometimes referred to as a “query frame”.  RuleMergeVirtual attempts to merge child frames into the parent frame.   The merge involves renaming any symbols in the lower frame that overlap with symbols in the upper frame.  It also involves merging the join information together.
- RuleRemoveOptionalJoins – removes optional join nodes form the plan tree as soon as possible so that planning will be more optimal.
- RulePlanJoins – this rule attempts to find an optimal ordering of the joins performed in the plan, while ensuring that dependencies are met.  This rule has three main steps.  First it must determine an ordering of joins that satisfy the 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.
- RuleCopyCriteria - 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).
- RuleCleanCriteria - this rule cleans up criteria after all the other rules.
- RuleMergeCriteria - looks for adjacent criteria nodes and merges them together.  It looks for adjacent identical conjuncts and removes duplicates.

- RuleRaiseAccess - 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.

- RuleChooseDependent - 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:

  1. there is at least one equi-join criterion, i.e. tablea.col = tableb.col

  2. 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:

  1. There is an unsatisfied access pattern that can be satisfied with the dependent join criteria

  2. The potential dependent side of the join is marked with an option makedep

  3. (4.3.2) if costing was enabled, the estimated cost for the dependent join (5.0+ 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.

  4. If key metadata information indicates that the potential dependent side is not "small" and the other side is "not small" or (5.0.1) 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.

- RuleChooseJoinStrategy – Determines the base join strategy. Currently this is a decision as to whether to use a merge join rather than the default strategy, which is a nested loop join. Ideally the choice of a hash join would also be evaluated here. Also costing should be used to determine the strategy cost.

- RuleCollapseSource - this rule removes all nodes below an Access node and collapses them into an equivalent query that is placed in the Access node.
- RuleAssignOutputElements - 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. 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.
- RuleValidateWhereAll - this rule validates a rarely used model option.
- RuleAccessPatternValidation – validates that all access patterns have been satisfied.

### 10.5.2. 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.

### 10.5.3. XML Planner

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

- 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.

# Architecture

## 11.1. Terminology

- VM or Process – a JBossAS instance running Teiid.
- 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. Active sessions are stored in a distributed cache and shared between Session services in each VM.
- Authorization – the Authorization service manages user entitlements. Entitlements use is optional (as specified in the configuration) and off by default.
- Buffer Manager – the *Buffer Manager* service provides access to data management for intermediate results.
- Transaction – the Transaction service manages global, local, and request scoped transactions. See also the documentation on *transaction support*.

## 11.2. Data Management

### 11.2.1. Cursoring and Batching

Teiid 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.

Teiid processes results in batches. A batch is simply a set of records. The number of rows in a batch is determined by the buffer system properties Processor Batch Size (within query engine) and Connector Batch Size (created at connectors).

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.

### 11.2.2. 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 max limit.

#### 11.2.2.1. Memory Management

The buffer manager has two storage managers - 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.

#### 11.2.2.2. 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. The file is random access. 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 whole.

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).

### 11.2.3. 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.

## 11.3. Query Termination

### 11.3.1. 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.

### 11.3.2. Timeouts

Timeouts in Teiid are managed on the client-side, in the JDBC API (which underlies both SOAP and ODBC access). 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 by the JDBC API without the client’s intervention.

## 11.4. Processing

### 11.4.1. 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.

Any of the Join Algorithms above can be made into a dependent join. The decision to implement a dependent join is considered after the join algorithm is chosen, and does not currently influence the algorithm selection.

### 11.4.2. 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 require all of the result set to ever 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 may be more than can fit in memory, this may result in 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. In this way, each pass reduces the number of sorted streams. When only one stream remains, it is the final output.

# Appendix A. BNF for SQL Grammar

## A.1. TOKENS

<DEFAULT> SKIP : { " " | "\t" | "\n" | "\r" }

<DEFAULT> MORE : { "/*" : IN_MULTI_LINE_COMMENT }

<IN_MULTI_LINE_COMMENT> SPECIAL : { <MULTI_LINE_COMMENT: "*/"> : DEFAULT }

<IN_MULTI_LINE_COMMENT> MORE : { <~[]> }

<DEFAULT> TOKEN : { <STRING: "string"> | <VARCHAR: "varchar"> | <BOOLEAN: "boolean"> | <BYTE: "byte"> | <TINYINT: "tinyint"> | <SHORT: "short"> | <SMALLINT: "smallint"> | <CHAR: "char"> | <INTEGER: "integer"> | <LONG: "long"> | <BIGINT: "bigint"> | <BIGINTEGER: "biginteger"> | <FLOAT: "float"> | <REAL: "real"> | <DOUBLE: "double"> | <BIGDECIMAL: "bigdecimal"> | <DECIMAL: "decimal"> | <DATE: "date"> | <TIME: "time"> | <TIMESTAMP: "timestamp"> | <OBJECT: "object"> | <BLOB: "blob"> | <CLOB: "clob"> | <XML: "xml"> }

<DEFAULT> TOKEN : { <CAST: "cast"> | <CONVERT: "convert"> }

<DEFAULT> TOKEN : { <ADD: "add"> | <ALL: "all"> | <ALTER: "alter"> | <AND: "and"> | <ANY: "any"> | <ARRAY: "array"> | <AS: "as"> | <ASC: "asc"> | <ATOMIC: "atomic"> | <AUTORIZATION: "authorization"> | <BEGIN: "begin"> | <BETWEEN: "between"> | <BINARY: "binary"> | <BOTH: "both"> | <BREAK: "break"> | <BY: "by"> | <CALL: "call"> | <CALLED: "called"> | <CASCADED: "cascaded"> | <CASE: "case"> | <CHARACTER: "character"> | <CHECK: "check"> | <CLOSE: "close"> | <COLLATE: "collate"> | <COLUMN: "column"> | <COMMIT: "commit"> | <CONNECT: "connect"> | <CONSTRAINT: "constraint"> | <CONTINUE: "continue"> | <CORRESPONDING: "corresponding"> | <CURRENT_DATE: "current_date"> | <CURRENT_TIME: "current_time"> | <CURRENT_TIMESTAMP: "current_timestamp"> | <CURRENT_USER: "current_user"> | <CREATE: "create"> | <CRITERIA: "criteria"> | <CROSS: "cross"> | <CURSOR: "cursor"> | <DAY: "day"> | <DEALLOCATE: "deallocate"> | <DEFAULT_KEYWORD: "default"> | <DECLARE: "declare"> | <DELETE: "delete"> | <DESC: "desc"> | <DESCRIBE: "describe"> | <DETERMINISTIC: "deterministic"> | <DISCONNECT: "disconnect"> | <DISTINCT: "distinct"> | <DROP: "drop"> | <ELSE: "else"> | <END: "end"> | <ERROR: "error"> | <ESCAPE: "escape"> | <EXCEPT: "except"> | <EXEC: "exec"> | <EXECUTE: "execute"> | <EXTERNAL: "external"> | <EXISTS: "exists"> | <FALSE: "false"> | <FETCH: "fetch"> | <FILTER: "filter"> | <FOR: "for"> | <FORIEGN: "foriegn"> | <FROM: "from"> | <FULL: "full"> | <FUNCTION: "function"> | <GET: "get"> | <GLOBAL: "global"> | <GRANT: "grant"> | <GROUP: "group"> | <HAS: "has"> | <HAVING: "having"> | <HOLD: "hold"> | <HOUR: "hour"> | <IF: "if"> | <IDENTITY: "identity"> | <IMMEDIATE: "immediate"> | <IN: "in"> | <INDICATOR: "indicator"> | <INNER: "inner"> | <INPUT: "input"> | <INOUT: "inout"> | <INSENSITIVE: "insensitive"> | <INSERT: "insert"> | <INTERSECT: "intersect"> | <INTERVAL: "interval"> | <INTO: "into"> | <IS: "is"> | <ISOLATION: "isolation"> | <JOIN: "join"> | <LEFT: "left"> | <LANGUAGE: "language"> | <LARGE: "large"> | <LEADING: "leading"> | <LIKE: "like"> | <LIMIT: "limit"> | <LOCAL: "local"> | <LOOP: "loop"> | <MAKEDEP: "makedep"> | <MAKENOTDEP: "makenotdep"> | <MATCH: "match"> | <MERGE: "merge">

| <METHOD: "method"> | <MINUTE: "minute"> | <MODIFIES: "modifies"> | <MODULE: "module"> | <MONTH: "month"> | <NATURAL: "natural"> | <NEW: "new"> | <NOCACHE: "nocache"> | <NO: "no"> | <NONE: "none"> | <NOT: "not"> | <NULL: "null"> | <OF: "of"> | <OLD: "old"> | <ON: "on"> | <ONLY: "only"> | <OPEN: "open"> | <OPTION: "option"> | <OR: "or"> | <ORDER: "order"> | <OUTER: "outer"> | <OUTPUT: "output"> | <OVER: "over"> | <OVERLAPS: "OVERLAPS"> | <PARAMETER: "parameter"> | <PARTITION: "partition"> | <PRECISION: "precision"> | <PREPARE: "prepare"> | <PRIMARY: "primary"> | <PROCEDURE: "procedure"> | <RANGE: "range"> | <READS: "reads"> | <RECURSIVE: "recursive"> | <REFERENCES: "REFERENCES"> | <REFERENCING: "REFERENCING"> | <RETURN: "return"> | <RETURNS: "returns"> | <REVOKE: "REVOKE"> | <RIGHT: "right"> | <ROLLBACK: "ROLLBACK"> | <ROLLUP: "ROLLUP"> | <ROW: "row"> | <ROWS: "rows"> | <SAVEPOINT: "savepoint"> | <SCROLL: "scroll"> | <SEARCH: "search"> | <SECOND: "second"> | <SELECT: "select"> | <SENSITIVE: "sensitive"> | <SESSION_USER: "session_user"> | <SET: "set"> | <SIMILAR: "similar"> | <SPECIFIC: "specific"> | <SOME: "some"> | <SQL: "sql"> | <SQLEXCEPTION: "sqlexception"> | <SQLSTATE: "sqlstate"> | <SQLWARNING: "sqlwarning"> | <START: "start"> | <STATIC: "static"> | <SYSTEM: "system"> | <SYSTEM_USER: "system_user"> | <TABLE: "table"> | <TEMPORARY: "temporary"> | <THEN: "then"> | <TIMEZONE_HOUR: "timezone_hour"> | <TIMEZONE_MINUTE: "timezone_minute"> | <TO: "to"> | <TRAILING: "trailing"> | <TRANSLATE: "translate"> | <TRIGGER: "trigger"> | <TRUE: "true"> | <UNION: "union"> | <UNIQUE: "unique"> | <UNKNOWN: "unknown"> | <USER: "user"> | <UPDATE: "update"> | <USING: "using"> | <VALUE: "value"> | <VALUES: "values"> | <VIRTUAL: "virtual"> | <WHEN: "when"> | <WHENEVER: "whenever"> | <WHERE: "where"> | <WITH: "with"> | <WHILE: "while"> | <WINDOW: "window"> | <WITHIN: "within"> | <WITHOUT: "without"> | <YEAR: "year"> | <ALLOCATE: "allocate"> | <ARE: "are"> | <ASENSITIVE: "asensitive"> | <ASYMETRIC: "asymetric"> | <CYCLE: "cycle"> | <DEC: "dec"> | <DEREF: "deref"> | <DYNAMIC: "dynamic"> | <ELEMENT: "element"> | <FREE: "free"> | <INT: "int"> | <LATERAL: "lateral"> | <LOCALTIME: "localtime"> | <LOCALTIMESTAMP: "localtimestamp"> | <MEMBER: "member"> | <MULTISET: "multiset"> | <NATIONAL: "national"> | <NCHAR: "nchar"> | <NCLOB: "nclob"> | <NUMERIC: "numeric"> | <RELEASE: "release"> | <SPECIFICTYPE: "specifictype"> | <SYMETRIC: "symetric"> | <SUBMULTILIST: "submultilist"> | <TRANSLATION: "translation"> | <TREAT: "treat"> | <VARYING: "varying"> }

<DEFAULT> TOKEN : { <XMLAGG: "xmlagg"> | <XMLATTRIBUTES: "xmlattributes"> | <XMLBINARY: "xmlbinary"> | <XMLCAST: "xmlcast"> | <XMLCONCAT: "xmlconcat"> | <XMLCOMMENT: "xmlcomment"> | <XMLDOCUMENT: "xmldocument"> | <XMLELEMENT: "xmlelement"> | <XMLEXISTS: "xmlexists"> | <XMLFOREST: "xmlforest"> | <XMLITERATE: "xmliterate"> | <XMLNAMESPACES: "xmlnamespaces"> | <XMLPARSE: "xmlparse"> | <XMLPI: "xmlpi"> | <XMLQUERY: "xmlquery"> | <XMLSERIALIZE: "xmlserialize"> | <XMLTABLE: "xmltable"> | <XMLTEXT: "xmltext"> | <XMLVALIDATE: "xmlvalidate"> }

<DEFAULT> TOKEN : { <DATALINK: "datalink"> | <DLNEWCOPY: "dlnewcopy"> | <DLPREVIOUSCOPY: "dlpreviouscopy"> | <DLURLCOMPLETE: "dlurlcomplete"> | <DLURLCOMPLETEWRITE: "dlurlcompletewrite"> | <DLURLCOMPLETEONLY: "dlurlcompleteonly"> | <DLURLPATH: "dlurlpath"> | <DLURLPATHWRITE: "dlurlpathwrite"> |

| |
|---|
| <DLURLPATHONLY: "dlurlpathonly"> \| <DLURLSCHEME: "dlurlscheme"> \| <DLURLSERVER: "dlurlserver"> \| <DLVALUE: "dlvalue"> \| <IMPORT: "import"> } |
| <DEFAULT> TOKEN : { <ALL_IN_GROUP: <ID> <PERIOD> <STAR>> \| <ID: <QUOTED_ID> (<PERIOD> <QUOTED_ID>)*> \| <#QUOTED_ID: <ID_PART> \| "\"" ("\"\"" \| ~["\""])+ "\""> \| <#ID_PART: ("@" \| "#" \| <LETTER>) (<LETTER> \| "_" \| <DIGIT>)*> \| <DATETYPE: "{" "d"> \| <TIMETYPE: "{" "t"> \| <TIMESTAMPTYPE: "{" "ts"> \| <BOOLEANTYPE: "{" "b"> \| <XMLTYPE: "{" "x"> \| <INTEGERVAL: (<MINUS>)? (<DIGIT>)+> \| <FLOATVAL: (<MINUS>)? (<DIGIT>)* <PERIOD> (<DIGIT>)+ (["e","E"] (["+","-"])? (<DIGIT>)+)?> \| <STRINGVAL: ("N")? "\'" ("\'\'" \| ~["\'"])* "\'"> \| <#LETTER: ["a"-"z","A"-"Z"] \| ["\u0153"-"\ufffd"]> \| <#DIGIT: ["0"-"9"]> } |
| <DEFAULT> TOKEN : { <COMMA: ","> \| <PERIOD: "."> \| <LPAREN: "("> \| <RPAREN: ")"> \| <LBRACE: "{"> \| <RBRACE: "}"> \| <EQ: "="> \| <NE: "<>"> \| <NE2: "!="> \| <LT: "<"> \| <LE: "<="> \| <GT: ">"> \| <GE: ">="> \| <STAR: "*"> \| <SLASH: "/"> \| <PLUS: "+"> \| <MINUS: "-"> \| <QMARK: "?"> \| <DOLLAR: "$"> \| <SEMICOLON: ";"> \| <CONCAT_OP: "\|\|"> } |

## A.2. NON-TERMINALS

| | |
|---|---|
| stringVal | ::= ( <STRINGVAL> ) |
| id | ::= ( <ID> ) |
| command | ::= ( *createUpdateProcedure* \| *userCommand* \| *callableStatement* ) ( <SEMICOLON> )? <EOF> |
| userCommand | ::= ( *queryExpression* \| *storedProcedure* \| *insert* \| *update* \| *delete* \| *dropTable* \| *createTempTable* ) |
| dropTable | ::= <DROP> <TABLE> *id* |
| createTempTable | ::= <CREATE> <LOCAL> <TEMPORARY> <TABLE> *id* <LPAREN> *createElementsWithTypes* <RPAREN> |
| errorStatement | ::= <ERROR> *expression* |
| statement | ::= ( *ifStatement* \| *loopStatement* \| *whileStatement* \| *delimitedStatement* ) |
| delimitedStatement | ::= ( *sqlStatement* \| *errorStatement* \| *assignStatement* \| *declareStatement* \| *continueStatement* \| *breakStatement* ) <SEMICOLON> |
| block | ::= ( *statement* \| ( <BEGIN> ( *statement* )* <END> ) ) |
| breakStatement | ::= <BREAK> |
| continueStatement | ::= <CONTINUE> |
| whileStatement | ::= <WHILE> <LPAREN> *criteria* <RPAREN> *block* |
| loopStatement | ::= <LOOP> <ON> <LPAREN> *queryExpression* <RPAREN> <AS> *id* *block* |
| ifStatement | ::= <IF> <LPAREN> *criteria* <RPAREN> *block* ( <ELSE> *block* )? |
| criteriaSelector | |

| | |
|---|---|
| | ::= ( ( <EQ> \| <NE> \| <NE2> \| <LE> \| <GE> \| <LT> \| <GT> \| <IN> \| <LIKE> \| ( <IS> <NULL> ) \| <BETWEEN> ) )? <CRITERIA> ( <ON> <LPAREN> *id* ( <COMMA> *id* )* <RPAREN> )? |
| hasCriteria | ::= <HAS> *criteriaSelector* |
| declareStatement | ::= <DECLARE> *dataType* *id* ( <EQ> *assignStatementOperand* )? |
| assignStatement | ::= *id* <EQ> *assignStatementOperand* |
| assignStatementOperand | ::= ( ( *insert* ) \| *update* \| *delete* \| *storedProcedure* \| ( *expression* ) \| *queryExpression* ) |
| sqlStatement | ::= ( ( *dynamicCommand* ) \| *userCommand* ) |
| translateCriteria | ::= <TRANSLATE> *criteriaSelector* ( <WITH> <LPAREN> *id* <EQ> *expression* ( <COMMA> *id* <EQ> *expression* )* <RPAREN> )? |
| createUpdateProcedure | ::= <CREATE> ( <VIRTUAL> )? ( <UPDATE> )? <PROCEDURE> *block* |
| dynamicCommand | ::= ( <EXECUTE> \| <EXEC> ) <STRING> *expression* ( <AS> *createElementsWithTypes* ( <INTO> *id* )? )? ( <USING> *setClauseList* )? ( <UPDATE> ( ( <INTEGERVAL> ) \| ( <STAR> ) ) )? |
| setClauseList | ::= *id* <EQ> ( <COMMA> *id* <EQ> )* |
| createElementsWithTypes | ::= *id* *dataType* ( <COMMA> *id* *dataType* )* |
| callableStatement | ::= <LBRACE> ( <QMARK> <EQ> )? <CALL> *id* ( <LPAREN> ( *executeUnnamedParams* ) <RPAREN> )? <RBRACE> ( *option* )? |
| storedProcedure | ::= ( ( ( <EXEC> ) \| ( <EXECUTE> ) \| ( <CALL> ) ) *id* <LPAREN> ( *executeNamedParams* \| *executeUnnamedParams* ) <RPAREN> ) ( *option* )? |
| executeUnnamedParams | ::= ( *expression* ( <COMMA> *expression* )* )? |
| executeNamedParams | ::= ( *id* <EQ> *expression* ( <COMMA> *id* <EQ> *expression* )* ) |
| insert | ::= <INSERT> <INTO> *id* ( <LPAREN> *id* ( <COMMA> *id* )* <RPAREN> )? ( ( <VALUES> *rowValues* ) \| ( *queryExpression* ) ) ( *option* )? |
| rowValues | ::= <LPAREN> *expression* ( <COMMA> *expression* )* <RPAREN> |
| update | ::= <UPDATE> *id* <SET> *setClauseList* ( *where* )? ( *option* )? |
| delete | ::= <DELETE> <FROM> *id* ( *where* )? ( *option* )? |
| queryExpression | ::= *queryExpressionBody* |
| queryExpressionBody | ::= *queryTerm* ( ( <UNION> \| <EXCEPT> ) ( <ALL> \| <DISTINCT> )? *queryTerm* )* ( *orderby* )? ( *limit* )? ( *option* )? |
| queryTerm | ::= *queryPrimary* ( <INTERSECT> ( <ALL> \| <DISTINCT> )? *queryPrimary* )* |
| queryPrimary | ::= ( *query* \| ( <LPAREN> *queryExpressionBody* <RPAREN> ) ) |
| query | ::= *select* ( *into* )? ( *from* ( *where* )? ( *groupBy* )? ( *having* )? )? |

| | |
|---|---|
| into | ::= <INTO> ( *id* ) |
| select | ::= <SELECT> ( <ALL> \| ( <DISTINCT> ) )? ( <STAR> \| ( *selectSymbol* ( <COMMA> *selectSymbol* )* ) ) |
| selectSymbol | ::= ( *selectExpression* \| *allInGroupSymbol* ) |
| selectExpression | ::= ( *expression* ( ( <AS> )? *id* )? ) |
| derivedColumn | ::= ( *expression* ( <AS> *id* )? ) |
| allInGroupSymbol | ::= <ALL_IN_GROUP> |
| xmlAgg | ::= <XMLAGG> <LPAREN> *expression* ( *orderby* )? <RPAREN> |
| aggregateSymbol | ::= ( ( *nonReserved* <LPAREN> <STAR> <RPAREN> ) \| ( *nonReserved* <LPAREN> ( <DISTINCT> )? *expression* <RPAREN> ) ) |
| from | ::= <FROM> ( *tableReference* ( <COMMA> *tableReference* )* ) |
| tableReference | ::= ( ( <LBRACE> *nonReserved* *tableReferenceUnescaped* <RBRACE> ) \| *tableReferenceUnescaped* ) |
| tableReferenceUnescaped | ::= ( *joinedTable* \| *tablePrimary* ) |
| joinedTable | ::= *tablePrimary* ( ( *crossJoin* \| *qualifiedJoin* ) )+ |
| crossJoin | ::= ( ( <CROSS> \| <UNION> ) <JOIN> *tablePrimary* ) |
| qualifiedJoin | ::= ( ( ( <RIGHT> ( <OUTER> )? ) \| ( <LEFT> ( <OUTER> )? ) \| ( <FULL> ( <OUTER> )? ) \| <INNER> )? <JOIN> *tableReference* <ON> *criteria* ) |
| tablePrimary | ::= ( *textTable* \| *xmlTable* \| *unaryFromClause* \| *subqueryFromClause* \| ( <LPAREN> *joinedTable* <RPAREN> ) ) ( ( <MAKEDEP> ) \| ( <MAKENOTDEP> ) )? |
| xmlSerialize | ::= <XMLSERIALIZE> <LPAREN> *nonReserved* *expression* ( <AS> ( <STRING> \| <VARCHAR> \| <CLOB> ) )? <RPAREN> |
| nonReserved | ::= <ID> |
| textTable | ::= <ID> <LPAREN> *expression* *nonReserved* *textColumn* ( <COMMA> *textColumn* )* ( <ID> *charVal* )? ( ( <ESCAPE> *charVal* ) \| ( <ID> *charVal* ) )? ( <ID> ( *intVal* )? )? ( <ID> *intVal* )? <RPAREN> ( <AS> )? *id* |
| textColumn | ::= *id* *dataType* ( <ID> *intVal* )? |
| xmlQuery | ::= <XMLQUERY> <LPAREN> ( *xmlNamespaces* <COMMA> )? *stringVal* ( <ID> *derivedColumn* ( <COMMA> *derivedColumn* )* )? ( ( <NULL> \| *nonReserved* ) <ON> *nonReserved* )? <RPAREN> |
| xmlTable | ::= <XMLTABLE> <LPAREN> ( *xmlNamespaces* <COMMA> )? *stringVal* ( <ID> *derivedColumn* ( <COMMA> *derivedColumn* )* )? ( <ID> *xmlColumn* ( <COMMA> *xmlColumn* )* )? <RPAREN> ( <AS> )? *id* |

| | |
|---|---|
| xmlColumn | ::= *id* ( ( &lt;FOR&gt; *nonReserved* ) \| ( *dataType* ( &lt;DEFAULT_KEYWORD&gt; *expression* )? ( *nonReserved* *stringVal* )? ) ) |
| intVal | ::= &lt;INTEGERVAL&gt; |
| subqueryFromClause | ::= ( &lt;TABLE&gt; )? &lt;LPAREN&gt; ( *queryExpression* \| *storedProcedure* ) &lt;RPAREN&gt; ( &lt;AS&gt; )? *id* |
| unaryFromClause | ::= ( &lt;ID&gt; ( ( &lt;AS&gt; )? *id* )? ) |
| where | ::= &lt;WHERE&gt; *criteria* |
| criteria | ::= *compoundCritOr* |
| compoundCritOr | ::= *compoundCritAnd* ( &lt;OR&gt; *compoundCritAnd* )* |
| compoundCritAnd | ::= *notCrit* ( &lt;AND&gt; *notCrit* )* |
| notCrit | ::= ( &lt;NOT&gt; )? *primary* |
| primary | ::= ( *predicate* \| ( &lt;LPAREN&gt; *criteria* &lt;RPAREN&gt; ) ) |
| predicate | ::= ( *subqueryCompareCriteria* \| *compareCrit* \| *matchCrit* \| *betweenCrit* \| *setCrit* \| *existsCriteria* \| *hasCriteria* \| *translateCriteria* \| *isNullCrit* ) |
| compareCrit | ::= *expression* ( &lt;EQ&gt; \| &lt;NE&gt; \| &lt;NE2&gt; \| &lt;LT&gt; \| &lt;LE&gt; \| &lt;GT&gt; \| &lt;GE&gt; ) *expression* |
| subquery | ::= &lt;LPAREN&gt; ( *queryExpression* \| *storedProcedure* ) &lt;RPAREN&gt; |
| subqueryCompareCriteria | ::= *expression* ( &lt;EQ&gt; \| &lt;NE&gt; \| &lt;NE2&gt; \| &lt;LT&gt; \| &lt;LE&gt; \| &lt;GT&gt; \| &lt;GE&gt; ) ( &lt;ANY&gt; \| &lt;SOME&gt; \| &lt;ALL&gt; ) *subquery* |
| matchCrit | ::= ( *expression* ( &lt;NOT&gt; )? &lt;LIKE&gt; *expression* ( &lt;ESCAPE&gt; *charVal* \| ( &lt;LBRACE&gt; &lt;ESCAPE&gt; *charVal* &lt;RBRACE&gt; ) )? ) |
| charVal | ::= *stringVal* |
| betweenCrit | ::= *expression* ( &lt;NOT&gt; )? &lt;BETWEEN&gt; *expression* &lt;AND&gt; *expression* |
| isNullCrit | ::= *expression* &lt;IS&gt; ( &lt;NOT&gt; )? &lt;NULL&gt; |
| setCrit | ::= *expression* ( &lt;NOT&gt; )? &lt;IN&gt; ( ( *subquery* ) \| ( &lt;LPAREN&gt; *expression* ( &lt;COMMA&gt; *expression* )* &lt;RPAREN&gt; ) ) |
| existsCriteria | ::= &lt;EXISTS&gt; *subquery* |
| groupBy | ::= &lt;GROUP&gt; &lt;BY&gt; ( *groupByItem* ( &lt;COMMA&gt; *groupByItem* )* ) |
| groupByItem | ::= *expression* |
| having | ::= &lt;HAVING&gt; *criteria* |
| orderby | ::= &lt;ORDER&gt; &lt;BY&gt; *sortKey* ( &lt;ASC&gt; \| &lt;DESC&gt; )? ( &lt;COMMA&gt; *sortKey* ( &lt;ASC&gt; \| &lt;DESC&gt; )? )* |
| sortKey | ::= *expression* |
| limit | ::= &lt;LIMIT&gt; ( &lt;INTEGERVAL&gt; \| &lt;QMARK&gt; ) ( &lt;COMMA&gt; ( &lt;INTEGERVAL&gt; \| &lt;QMARK&gt; ) )? |

| | |
|---|---|
| option | ::= <OPTION> ( <MAKEDEP> id ( <COMMA> id )* \| <MAKENOTDEP> id ( <COMMA> id )* \| <NOCACHE> ( id ( <COMMA> id )* )? )* |
| expression | ::= concatExpression |
| concatExpression | ::= ( plusExpression ( <CONCAT_OP> plusExpression )* ) |
| plusExpression | ::= ( timesExpression ( plusOperator timesExpression )* ) |
| plusOperator | ::= ( <PLUS> \| <MINUS> ) |
| timesExpression | ::= ( basicExpression ( timesOperator basicExpression )* ) |
| timesOperator | ::= ( <STAR> \| <SLASH> ) |
| basicExpression | ::= ( <QMARK> \| literal \| ( <LBRACE> nonReserved function <RBRACE> ) \| ( aggregateSymbol ) \| ( xmlAgg ) \| ( function ) \| ( <ID> ) \| ( <LPAREN> expression <RPAREN> ) \| subquery \| caseExpression \| searchedCaseExpression ) |
| caseExpression | ::= <CASE> expression ( <WHEN> expression <THEN> expression )+ ( <ELSE> expression )? <END> |
| searchedCaseExpression | ::= <CASE> ( <WHEN> criteria <THEN> expression )+ ( <ELSE> expression )? <END> |
| function | ::= ( ( <CONVERT> <LPAREN> expression <COMMA> dataType <RPAREN> ) \| ( <CAST> <LPAREN> expression <AS> dataType <RPAREN> ) \| ( nonReserved <LPAREN> intervalType <COMMA> expression <COMMA> expression <RPAREN> ) \| queryString \| ( ( <LEFT> \| <RIGHT> \| <CHAR> \| <USER> \| <YEAR> \| <MONTH> \| <HOUR> \| <MINUTE> \| <SECOND> \| <XMLCONCAT> \| <XMLCOMMENT> ) <LPAREN> ( expression ( <COMMA> expression )* )? <RPAREN> ) \| ( ( <INSERT> ) <LPAREN> ( expression ( <COMMA> expression )* )? <RPAREN> ) \| ( ( <TRANSLATE> ) <LPAREN> ( expression ( <COMMA> expression )* )? <RPAREN> ) \| xmlParse \| xmlElement \| ( <XMLPI> <LPAREN> ( <ID> idExpression \| idExpression ) ( <COMMA> expression )? <RPAREN> ) \| xmlForest \| xmlSerialize \| xmlQuery \| ( id <LPAREN> ( expression ( <COMMA> expression )* )? <RPAREN> ) ) |
| xmlParse | ::= <XMLPARSE> <LPAREN> nonReserved expression ( nonReserved )? <RPAREN> |
| queryString | ::= nonReserved <LPAREN> expression ( <COMMA> derivedColumn )* <RPAREN> |
| xmlElement | ::= <XMLELEMENT> <LPAREN> ( <ID> id \| id ) ( <COMMA> xmlNamespaces )? ( <COMMA> xmlAttributes )? ( <COMMA> expression )* <RPAREN> |
| xmlAttributes | ::= <XMLATTRIBUTES> <LPAREN> derivedColumn ( <COMMA> derivedColumn )* <RPAREN> |

| | |
|---|---|
| xmlForest | ::= <XMLFOREST> <LPAREN> ( *xmlNamespaces* <COMMA> )? *derivedColumn* ( <COMMA> *derivedColumn* )* <RPAREN> |
| xmlNamespaces | ::= <XMLNAMESPACES> <LPAREN> *namespaceItem* ( <COMMA> *namespaceItem* )* <RPAREN> |
| namespaceItem | ::= ( *stringVal* <AS> *id* ) |
| | ::= ( <NO> <DEFAULT_KEYWORD> ) |
| | ::= ( <DEFAULT_KEYWORD> *stringVal* ) |
| idExpression | ::= *id* |
| dataType | ::= ( <STRING> \| <VARCHAR> \| <BOOLEAN> \| <BYTE> \| <TINYINT> \| <SHORT> \| <SMALLINT> \| <CHAR> \| <INTEGER> \| <LONG> \| <BIGINT> \| <BIGINTEGER> \| <FLOAT> \| <REAL> \| <DOUBLE> \| <BIGDECIMAL> \| <DECIMAL> \| <DATE> \| <TIME> \| <TIMESTAMP> \| <OBJECT> \| <BLOB> \| <CLOB> \| <XML> ) |
| intervalType | ::= ( *nonReserved* ) |
| literal | ::= ( *stringVal* \| <INTEGERVAL> \| <FLOATVAL> \| <FALSE> \| <TRUE> \| <UNKNOWN> \| <NULL> \| ( ( <BOOLEANTYPE> \| <TIMESTAMPTYPE> \| <DATETYPE> \| <TIMETYPE> \| <XMLTYPE> ) *stringVal* <RBRACE> ) ) |