Teiid - Scalable Information Integration

1

Teiid Reference Documentation

7.6

Preface ix
1. DML Support 1
1.1. Identifiers 1
1.1.1. Reserved Words 2
1.2. Expressions 2
1.2.1. Column Identifiers 3
1.2.2. Literals 3
1.2.3. Aggregate Functions 4
1.2.4. Window functions 5
1.2.5. Case and searched case 7
1.2.6. Scalar subqueries 7
1.2.7. Parameter references 7
1.3. Criteria 7
1.4. SQL Commands 9
1.4.1. SELECT Command 10
1.4.2. INSERT Command 11
1.4.3. UPDATE Command 11
1.4.4. DELETE Command 11
1.4.5. EXECUTE Command 12
1.4.6. Procedural Relational Command 12
1.5. Set Operations 13
1.6. Subqueries 14
1.6.1. Inline views 14
1.6.2. Subqueries can appear anywhere where an expression or criteria is expected. 14
2. SQL Clauses 17
2.1. WITH Clause 17
2.2. SELECT Clause 17
2.3. FROM Clause 17
2.3.1. From Clause Hints 18
2.3.2. Nested Table Reference 18
2.3.3. TEXTTABLE 19
2.3.4. XMLTABLE 21
2.4. ARRAYTABLE 22
2.5. WHERE Clause 23
2.6. GROUP BY Clause 23
2.7. HAVING Clause 24
2.8. ORDER BY Clause 24
2.9. LIMIT Clause 25
2.10. INTO Clause 26
2.11. OPTION Clause 26
3. DDL Support 29
3.1. Temp Tables 29
3.2. Alter View 30

3.3. Alter Procedure 31
3.4. Create Trigger 31
3.5. Alter Trigger 31
4. XML SELECT Command 33
4.1. Overview 33
4.2. Query Structure 33
4.2.1. FROM Clause 33
4.2.2. SELECT Clause 33
4.2.3. WHERE Clause 34
4.2.4. ORDER BY Clause 36
4.3. Document Generation 36
4.3.1. Document Validation 37
5. Datatypes 39
5.1. Supported Types 39
5.2. Type Conversions 41
5.3. Special Conversion Cases 42
5.3.1. Conversion of String Literals 42
5.3.2. Converting to Boolean 43
5.3.3. Date/Time/Timestamp Type Conversions 43
5.4. Escaped Literal Syntax 43
6. Scalar Functions 45
6.1. Numeric Functions 45
6.1.1. Parsing Numeric Datatypes from Strings 47
6.1.2. Formatting Numeric Datatypes as Strings 48
6.2. String Functions 48
6.3. Date/Time Functions 51
6.3.1. Parsing Date Datatypes from Strings 55
6.3.2. Specifying Time Zones 55
6.4. Type Conversion Functions 55
6.5. Choice Functions 56
6.6. Decode Functions 56
6.7. Lookup Function 58
6.8. System Functions 59
6.8.1. COMMANDPAYLOAD 59
6.8.2. ENV 60
6.8.3. SESSION_ID 60
6.8.4. USER 60
6.8.5. CURRENT_DATABASE 60
6.9. XML Functions 60
6.9.1. JSONTOXML 60
6.9.2. XMLCOMMENT 62
6.9.3. XMLCONCAT 62
6.9.4. XMLELEMENT 62
6.9.5. XMLFOREST 63

6.9.6. XMLPARSE 63
6.9.7. XMLPI 63
6.9.8. XMLQUERY 63
6.9.9. XMLSERIALIZE 64
6.9.10. XSLTRANSFORM 64
6.9.11. XPATHVALUE 65
6.10. Security Functions 65
6.10.1. HASROLE 65
6.11. Miscellaneous Functions 66
6.11.1. array_get 66
6.11.2. array_length 66
6.11.3. uuid 66
6.12. Nondeterministic Function Handling 66
7. Updatable Views 69
7.1. Key-preserved Table 69
8. Procedures 71
8.1. Procedure Language 71
8.1.1. Command Statement 71
8.1.2. Dynamic SQL Command 71
8.1.3. Declaration Statement 74
8.1.4. Assignment Statement 75
8.1.5. Compound Statement 75
8.1.6. If Statement 76
8.1.7. Loop Statement 77
8.1.8. While Statement 77
8.1.9. Continue Statement 77
8.1.10. Break Statement 78
8.1.11. Leave Statement 78
8.1.12. Error Statement 78
8.2. Virtual Procedures 79
8.2.1. Virtual Procedure Definition 79
8.2.2. Procedure Parameters 79
8.2.3. Example Virtual Procedures 80
8.2.4. Executing Virtual Procedures 81
8.2.5. Limitations 81
8.3. Update Procedures 81
8.3.1. Update Procedure Processing 82
8.3.2. For Each Row 82
9. Transaction Support 85
9.1. AutoCommitTxn Execution Property 85
9.2. Updating Model Count 86
9.3. JDBC and Transactions 86
9.3.1. JDBC API Functionality 86
9.3.2. J2EE Usage Models 86

9.4. Transactional Behavior with JBoss Data Source Types 87
9.5. Limitations and Workarounds 89
10. Data Roles 91
10.1. Permissions 91
10.2. Role Mapping 93
10.3. XML Definition 93
10.4. System Functions 95
10.5. Customizing 95
11. System Schema 97
11.1. System Tables 97
11.1.1. VDB, Schema, and Properties 97
11.1.2. Table Metadata 98
11.1.3. Procedure Metadata 101
11.1.4. Datatype Metadata 102
11.2. System Procedures 103
11.2.1. SYS.getXMLSchemas 103
11.2.2. SYSADMIN.logMsg 103
11.2.3. SYSADMIN.isLoggable 103
11.2.4. SYSADMIN.refreshMatView 104
11.2.5. SYSADMIN.refreshMatViewRow 104
11.2.6. Metadata Procedures 104
12. VDBs 107
12.1. VDB Definition 107
12.1.1. VDB Element 108
12.1.2. Model Element 108
12.1.3. Translator Element 109
12.2. Dynamic VDBs 109
12.3. Multi-Source Models and VDB 110
12.3.1. Multi-source SELECTs 111
12.3.2. Multi-source INSERTs 112
12.3.3. Multi-source UPDATEs 112
12.3.4. Multi-source DELETEs 112
12.3.5. Multi-source Stored Procedures 112
12.3.6. Additional Concerns 112
13. Translators 113
13.1. Introduction to the Teiid Connector Architecture 113
13.2. Translators 113
13.2.1. File Translator 114
13.2.2. JDBC Translator 116
13.2.3. LDAP Translator 121
13.2.4. Loopback Translator 122
13.2.5. Salesforce Translator 122
13.2.6. Web Services Translator 127
13.2.7. OLAP Translator 129

13.2.8. Delegating Translators 130
14. Federated Planning 131
14.1. Overview 131
14.2. Federated Optimizations 133
14.2.1. Access Patterns 133
14.2.2. Pushdown 133
14.2.3. Dependent Joins 133
14.2.4. Copy Criteria 134
14.2.5. Projection Minimization 134
14.2.6. Partial Aggregate Pushdown 134
14.2.7. Optional Join 134
14.2.8. Partitioned Union 136
14.2.9. Standard Relational Techniques 136
14.3. Subquery optimization 137
14.4. XQuery Optimization 138
14.5. Federated Failure Modes 139
14.5.1. Partial Results 139
14.6. Query Plans 139
14.6.1. Getting a Query Plan 140
14.6.2. Analyzing a Query Plan 140
14.6.3. Relational Plans 141
14.6.4. Source Hints 142
14.7. Query Planner 142
14.7.1. Relational Planner 143
14.7.2. Procedure Planner 148
14.7.3. XML Planner 148
15. Architecture 151
15.1. Terminology 151
15.2. Data Management 151
15.2.1. Cursoring and Batching 151
15.2.2. Buffer Management 151
15.2.3. Cleanup 152
15.3. Query Termination 152
15.3.1. Canceling Queries 152
15.3.2. User Query Timeouts 152
15.4. Processing 153
15.4.1. Join Algorithms 153
15.4.2. Sort Based Algorithms 153
A. BNF for SQL Grammar 155
A.1. TOKENS 155
A.2. NON-TERMINALS 157

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.

DML 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.

1.1.1. Reserved Words

Teiid's reserved words include the standard SQL 2003 Foundation, SQL/MED, and SQL/XML reserved words, as well as Teiid specific words such as BIGINTEGER, BIGDECIMAL, or MAKEDEP. See the *Appendix A, BNF for SQL Grammar* TOKENS section for all reserved words. They will appear as 'SMALLINT: "smallint"' where the quoted string is the actual lexical form.

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*

- *Window 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(x) – count the number of values (excluding nulls) in a group
- SUM(x) – sum of the values (excluding nulls) in a group
- AVG(x) – average of the values (excluding nulls) in a group
- MIN(x) – minimum value in a group (excluding null)
- MAX(x) – maximum value in a group (excluding null)
- ANY(x)/SOME(x) – returns TRUE if any value in the group is TRUE (excluding null)
- EVERY(x) – returns TRUE if every value in the group is TRUE (excluding null)
- VAR_POP(x) – biased variance (excluding null) logically equals (sum(x^2) - sum(x)^2/count(x))/count(x); returns a double; null if count = 0
- VAR_SAMP(x) – sample variance (excluding null) logically equals (sum(x^2) - sum(x)^2/count(x))/(count(x) - 1); returns a double; null if count < 2
- STDDEV_POP(x) – standard deviation (excluding null) logically equals SQRT(VAR_POP(x))
- STDDEV_SAMP(x) – sample standar deviation (excluding null) logically equals SQRT(VAR_SAMP(x))
- TEXTAGG(FOR (expression [as name], ... [DELIMITER char] [QUOTE char] [HEADER] [ENCODING id] *[ORDER BY ...]*) – CSV text aggregation of all expressions in each row of a group. When DELIMITER is not specified, by default comma(,) is used as delimiter. Double quotes(") is the default quote character. Use QUOTE to specify a differnt value. All non-null values will be quoted. If HEADER is specified, the result contains the header row as the first line - the header line will be present even if there are no rows in a group. This aggregation returns a blob.
- XMLAGG(xml_expr *[ORDER BY ...]*) – xml concatination of all xml expressions in a group (excluding null). The ORDER BY clause cannot reference alias names or use positional ordering.

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 cannot be used in FROM, GROUP BY, or WHERE clauses without an intervening query expression.
- Aggregate functions cannot be nested within another aggregate function without an intervening query expression.
- Aggregate functions may be nested inside other functions.
- Any aggregate function may take an optional FILTER clasue of the form

```
FILTER ( WHERE condition )
```

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

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

1.2.4. Window functions

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

Usage:

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

aggregate can be any *Section 1.2.3, "Aggregate Functions"*. Ranking can be one of ROW_NUMBER(), RANK(), DENSE_RANK().

Syntax Rules:

- Window functions can only appear in the SELECT and ORDER BY clauses of a query expression.
- Window functions cannot be nested in one another.
- Partitioning and order by expressions cannot contain subqueries or outer references.
- The ranking (ROW_NUMBER, RANK, DENSE_RANK) functions require the use of the window specification ORDER BY clause.

- An XMLAGG ORDER BY clause cannot be used when windowed.
- The window specification ORDER BY clause cannot reference alias names or use positional ordering.
- Windowed aggregates may not use DISTINCT if the window specification is ordered.

1.2.4.1. Function Definitions

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

1.2.4.2. Processing

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

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

Example 1.1. Example Windowed Results

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

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

name	salary	max_sal	rank	dense_rank	row_num
Suzie	60000	150000	3	3	4
Suzie	150000	150000	1	1	1

1.2.5. 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.

1.2.6. 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.

1.2.7. 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 may be:

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

Usage:

- `criteria AND|OR criteria`
- `NOT criteria`
- `(criteria)`
- `expression (=|<>|!=|<|>|<=|>=) (expression|((ANY|ALL|SOME) subquery))`
- `expression [NOT] IS NULL`
- `expression [NOT] IN (expression[,expression]*)|subquery`
- `expression [NOT] LIKE pattern [ESCAPE char]`

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

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

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

  Note

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

- `expression [NOT] LIKE_REGEX pattern`

  LIKE_REGEX allows for standard regular expression syntax to be used for matching. This differs from SIMILAR TO and LIKE in that the escape character is no longer used (\ is already the standard escape mechansim in regular expressions and % and _ have no special meaning. The runtime engine uses the JRE implementation of regular expressions - see the *java.util.regex.Pattern* [http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html] class for details.

Note

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

- ```
  EXISTS(subquery)
  ```
- ```
  expression [NOT] BETWEEN minExpression AND maxExpression
  ```

  Teiid converts BETWEEN into the equivalent form expression >= minExpression AND expression <= maxExpression
- ```
  expression
  ```

  Where expression has type boolean.

Syntax Rules:

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

Some examples of valid criteria are:

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

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:

- *WITH ...*
- *SELECT ...*
- *[FROM ...]*
- *[WHERE ...]*
- *[GROUP BY ...]*
- *[HAVING ...]*
- *[ORDER BY ...]*
- *[(LIMIT ...) | ([OFFSET ...] [FETCH ...])]*
- *[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.

- WITH stage - gathers all rows from all with items in the order listed. Subsequent with items and the main query can reference the a with item as if it is a table.
- FROM stage - gathers all rows from all tables involved in the query and logically joins them with a Cartesian product, producing a single large table with all columns from all tables. Joins and join criteria are then applied to filter rows that do not match the join structure.
- WHERE stage - applies a criteria to every output row from the FROM stage, further reducing the number of rows.
- GROUP BY stage - groups sets of rows with matching values in the group by columns.
- HAVING stage - applies criteria to each group of rows. Criteria can only be applied to columns that will have constant values within a group (those in the grouping columns or aggregate functions applied across the group).

- SELECT stage - specifies the column expressions that should be returned from the query. Expressions are evaluated, including aggregate functions based on the groups of rows, which will no longer exist after this point. The output columns are named using either column aliases or an implicit name determined by the engine. If SELECT DISTINCT is specified, duplicate removal will be performed on the rows being returned from the SELECT stage.
- ORDER BY stage - sorts the rows returned from the SELECT stage as desired. Supports sorting on multiple columns in specified order, ascending or descending. The output columns will be identical to those columns returned from the SELECT stage and will have the same name.
- LIMIT stage - returns only the specified rows (with skip and limit values).

This model 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.

Note

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

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. 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. Set Operations

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

Usage:

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

Syntax Rules:

- The output columns will be named by the output columns of the first set operation branch.

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

1.6. 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.

1.6.1. 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. An inline view is nearly identical to a traditional view. See also *Section 2.1, “WITH Clause”*.

Example 1.2. 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
```

1.6.2. Subqueries can appear anywhere where an expression or criteria is expected.

Subqueries are supported in quantified criteria, the EXISTS predicate, the IN predicate, and as *Section 1.2.6, “Scalar subqueries”*.

Example 1.3. Example Subquery in WHERE Using EXISTS

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

Subqueries can appear anywhere where an expression or criteria is expected.

Example 1.4. Example Quantified Comparison Subqueries

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

Example 1.5. Example IN Subquery

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

See also *Section 14.3, “Subquery optimization”*.

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.

2.1. WITH Clause

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

Usage:

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

Syntax Rules:

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

2.2. SELECT Clause

SQL queries that 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.

2.3. 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 /*+ MAKEDEP */ table2 ON join-criteria
- FROM table1 JOIN /*+ MAKENOTDEP */ table2 ON join-criteria
- FROM /*+ MAKEIND */ table1 JOIN table2 ON join-criteria
- FROM /*+ NO_UNNEST */ vw1 JOIN table2 ON join-criteria
- FROM table1 left outer join */*+ optional */* table2 ON join-criteria
- FROM *TEXTTABLE...*
- FROM *XMLTABLE...*
- FROM *ARRAYTABLE...*
- FROM *(SELECT ...*

2.3.1. From Clause Hints

MAKEIND, 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. The hints may appear in a comment that proceeds the from clause. The hints can be specified against any from clause, not just a named table.

NO_UNNEST can be specified against a subquery from clause or view to instruct the planner to not merge the nested SQL in the surrounding query - also known as view flattening. This hint only applies to Teiid planning and is not passed to source queries. NO_UNNEST may appear in a comment that proceeds the from clause.

2.3.2. 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.

2.3.3. 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>, ... [NO ROW DELIMITER] [DELIMITER
 char] [(QUOTE|ESCAPE) char] [HEADER [integer]] [SKIP integer]) AS name
```

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

Parameters

- expression - the text content to process, which should be convertable to CLOB.
- NO ROW DELIMITER indicates that fixed parsing should not assume the presense of newline row delimiters.
- DELIMITER sets the field delimiter character to use. Defaults to ','.

- QUOTE sets the quote, or qualifier, character used to wrap field values. Defaults to "".
- ESCAPE sets the escape character to use if no quoting character is in use. This is used in situations where the delimiter or new line characters are escaped with a preceding character, e.g. \,
- HEADER specifies the text line number (counting every new line) on which the column names occur. All lines prior to the header will be skipped. If HEADER is specified, then the header line will be used to determine the TEXTTABLE column position by case-insensitive name matching. This is especially useful in situations where only a subset of the columns are needed. If the HEADER value is not specified, it defaults to 1. If HEADER is not specified, then columns are expected to match positionally with the text contents.
- SKIP specifies the number of text lines (counting every new line) to skip before parsing the contents. HEADER may still be specified with SKP.
- WIDTH indicates the fixed-width length of a column in characters - not bytes. The CR NL newline value counts as a single character.
- NO TRIM specifies that the text value should not be trimmed of all leading and trailing white space.

Syntax Rules:

- If width is specified for one column it must be specified for all columns and be a non-negative integer.
- If width is specified, then fixed width parsing is used and ESCAPE, QUOTE, and HEADER should not be specified.
- If width is not specified, then NO ROW DELIMITER cannot be used.
- The columns names must be not contain duplicates.

Examples

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

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

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

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

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

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

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

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

- As a nested table:

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

2.3.4. 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 [62]*.

See XMLQUERY for the definition of *PASSING [64]*.

See also *XMLQUERY*

Note

See also *Section 14.4, "XQuery Optimization"*

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.
- A FOR ORDINALITY column is typed as integer and will return the 1-based item number as its value.
- Each non-ordinality column specifies a type and optionally a PATH and a DEFAULT expression.
- If PATH is not specified, then the path will be the same as the column name.

Syntax Rules:

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

Examples

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

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

- As a nested table:

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

2.4. ARRAYTABLE

The ARRAYTABLE funciton processes an array input to produce tabular ouptut. The function itself defines what columns it projects. The ARRAYTABLE function is implicitly a nested table and may be correlated to preceeding FROM clause entries.

Usage:

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

```
COLUMN := name datatype
```

Parameters

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

Syntax Rules:

- The columns names must be not contain duplicates.

Examples

- As a nested table:

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

ARRAYTABLE is effectively a shortcut for using the *Section 6.11.1, "array_get"* function in a nested table. For example "ARRAYGET(val COLUMNS col1 string, col2 integer) AS X" is the same as "TABLE(SELECT cast(array_get(val, 1) AS string) AS col1, cast(array_get(val, 2) AS integer) AS col2) AS X".

2.5. 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*

2.6. 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.

2.7. 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.

2.8. ORDER BY Clause

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

Usage:

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

Syntax Rules:

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

which is Teiid's internal default sorting behavior. However not all sources return results with nulss sorted as low values by default, and Teiid may return results with different null orderings.

Warning

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

2.9. LIMIT Clause

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

Usage:

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

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

Syntax Rules:

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

Examples:

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

- FETCH FIRST ROW ONLY - returns only the first record

2.10. 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.

2.11. 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

All tables specified in the OPTION clause should be fully qualified.

Note

Previous versions of Teiid accepted the PLANONLY, DEBUG, and SHOWPLAN option arguments. These are no longer accepted in the OPTION clause. Please see the Client Developers Guide for replacements to those options.

DDL Support

Teiid supports a subset of DDL to, create/drop temporary tables and to manipulate procedure and view definitions at runtime. It is not currently possible to arbitrarily drop/create non-temporary metadata entries.

Note

A `MetadataRepository` must be configured to make a non-temporary metadata update persistent. See the Developers Guide Runtime Metadata Updates section for more.

3.1. 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 INSERT statement or explicitly with a CREATE TABLE statement. Implicitly created temp tables must have a name that starts with '#'.

Creation syntax:

- Explicit: CREATE LOCAL TEMPORARY TABLE x (column type [NOT NULL], ... [PRIMARY KEY (column, ...)])
- Implicit: INSERT INTO #x (column, ...) VALUES (value, ...)

  If #x doen't exist, it will be defined using the given column names and types from the value expressions.
- Implicit: INSERT INTO #x [(column, ...)] select c1, c2 from t

  If #x doesn't exist, it will be defined using the target column names (in not supplied, the column names will match the derived column names from the query), and the types from the query derived columns.
- Use the SERIAL data type to specify a NOT NULL and auto-incrementing INTEGER column. The starting value of a SERIAL column is 1.

Drop syntax:

- DROP TABLE x

Primary Key Support

- All key columns must be comparable.

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

Limitations:

- With the CREATE TABLE syntax only basic table definition (column name and type information) and an optional primary key are supported.
- The "ON COMMIT" clause is not supported in the CREATE TABLE statement.
- "drop behavior" option is not supported in the drop statement.
- Only local temporary tables are supported. This implies that the scope of temp table will be either to the session or the block of a virtual procedure that creates it.
- Session level temp tables are not fail-over safe.
- Temp tables support a READ_UNCOMMITED transaction isolation level. There are no locking mechanisms available to support higher isolation levels and the result of a rollback may be inconsistent across multiple transactions. If concurrent transactions are not associated with the same local temporary table or session, then the transaction isolation level is effectively SERIALIZABLE. If you want full constency with local temporary tables, then only use a connection with 1 transaction at a time. This mode of operation is ensured by connection pooling that tracking connections by transaction.
- Lob values (xml, clob, blob) are tracked by reference rather than by value in a temporary table. Lob values from external sources that are inserted in a temporary table may become unreadable when the associated statement or connection is closed.

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

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

See *virtual procedures* for more on temp table usage.

3.2. Alter View

Usage:

```
ALTER VIEW name AS queryExpression
```

Syntax Rules:

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

3.3. Alter Procedure

Usage:

```
ALTER PROCEDURE name AS block
```

Syntax Rules:

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

3.4. Create Trigger

Usage:

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

Syntax Rules:

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

Limitations:

- There is no corresponding drop operation. See *Section 3.5, "Alter Trigger"* for enabling/disabling an existing trigger.

3.5. Alter Trigger

Usage:

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

Syntax Rules:

- The target, name, must be an updatable view.
- Triggers are not yet true schema objects. They are scoped only to their view and have no name.
- An *Section 8.3, "Update Procedures"* must already exist for the given trigger event.

Note

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

XML SELECT Command

4.1. 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.

4.2. 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.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

4.2.3.1. 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.

4.2.3.1.1. 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.

4.2.3.1.2. 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.

4.2.3.1.3. 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.

4.2.4. 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.

4.3. 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.

4.3.1. 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

5.1. 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 5.1. 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 or serial	numeric, integral type, signed 32-bit. The serial type also implies not null and has an auto-incrementing value that starts at 1. serial types are not automatically UNIQUE.	java.lang.Integer	INTEGER	INTEGER
long or bigint	numeric, integral type, signed 64-bit	java.lang.Long	BIGINT	NUMERIC
biginteger	numeric, integral type, arbitrary precision of up to 1000 digits	java.lang.BigInteger	NUMERIC	NUMERIC
float or real		java.lang.Float	REAL	FLOAT

Type	Description	Java Runtime Class	JDBC Type	ODBC Type
	numeric, floating point type, 32-bit IEEE 754 floating-point numbers			
double	numeric, floating point type, 64-bit IEEE 754 floating-point numbers	java.lang.String	DOUBLE	DOUBLE
bigdecimal or decimal	numeric, floating point type, arbitrary precision of up to 1000 digits. Note that the precision and scale cannot be explicitly set with the type literal, e.g. decimal(38, 2).	java.math.BigDecimal	NUMERIC	NUMERIC
date	datetime, representing a single day (year, month, day)	java.sql.Date	DATE	DATE
time	datetime, representing a single time (hours, minutes, seconds, milliseconds)	java.sql.Time	TIME	TIME
timestamp	datetime, representing a single date and time (year, month, day, hours, minutes, seconds, milliseconds, nanoseconds)	java.sql.Timestamp	TIMESTAMP	TIMESTAMP
object	any arbitrary Java object, must implement java.lang.Serializable	Any	JAVA_OBJECT	VARCHAR
blob	binary large object, representing a stream of bytes	java.sql.Blob [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

5.2. 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 5.2. Type Conversions

Source Type	Valid Implicit Target Types	Valid Explicit Target Types
string	clob	

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

[a] string to xml is equivlant to XMLPARSE(DOCUMENT exp) - See also *XMLPARSE*

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

5.3. Special Conversion Cases

5.3.1. 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.

5.3.2. 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
	other	true
Numeric	0	false
	other	true

5.3.3. 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` .

5.4. 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 5.3. 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.

6.1. 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

[a]The precision and scale of non-bigdecimal arithmetic function functions results matches that of Java. The results of bigdecimal operations match Java, except for division, which uses a preferred scale of max(16, dividend.scale + divisor.precision + 1), which then has trailing zeros removed by setting the scale to max(dividend.scale, normalized scale)

6.1.1. 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
'$25.30'	parseDouble(cost, '$#,##0.00;($#,##0.00)')	25.3	double
'25%'	parseFloat(percent, '#,##0%')	25	float
'2,534.1'	parseFloat(total, '#,##0.###;-#,##0.###')	2534.1	float
'1.234E3'	parseLong(amt, '0.###E0')	1234	long
'1,234,567'	parseInteger(total, '#,##0;-#,##0')	1234567	integer

6.1.2. 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'

6.2. 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 {integer}, 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 blank chars	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 ''. Names are optional for column reference expressions. e.g. QUERYSTRING('path', 'value' as "&x", ' & ' as y, null as z) returns 'path?%26x=value&y=%20%26%20'	path, expr in {string}. name is an identifier
REPEAT(str1,instances)	Repeat string1 a specified number of times	str1 in {string}, instances in {integer} return string
REPLACE(x, y, z)	Replace all y in x with z	x,y,z in {string}, return string
RIGHT(x, y)	Get right y characters of x	x in {string}, y in {integer}, return string
RPAD(input string x, pad length y)	Pad input string x with spaces on the right to the length of y	x in {string}, y in {integer}, return string
RPAD(x, y, z)	Pad input string x on the right to the length of y using character z	x in {string}, y in {string}, z in {character}, return string
RTRIM(x)	Right trim x of blank chars	x is string, return string
SUBSTRING(x, y) SUBSTRING(x FROM y)	Get substring from x, from position y to the end of x	y in {integer}
SUBSTRING(x, y, z) SUBSTRING(x FROM y FOR z)	Get substring from x from position y with length z	y, z in {integer}
TO_CHARS(x, encoding)	Return a clob from the blob with the given encoding. BASE64, HEX, and the 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

Function	Definition	Datatype Constraint
TRANSLATE(x, y, z)	Translate string x by replacing each character in y with the character in z at the same position	x in {string}
TRIM([[LEADING|TRAILING|BOTH] [x] FROM] y)	Trim the leading, trailing, or both ends of a string y of character x. If LEADING/TRAILING/BOTH is not specified, BOTH is used. If no trim character x is specficed then the blank space ' ' is used.	x in {character}, y in {string}
UCASE(x)	Uppercase of x	x in {string}
UNESCAPE(x)	Unescaped version of x. Possible escape sequences are \b - backspace, \t - tab, \n - line feed, \f - form feed, \r - carriage return. \uXXXX, where X is a hex value, can be used to specify any unicode character. \XXX, where X is an octal digit, can be used to specify an octal byte value. If any other character appears after an escape character, that character will appear in the output and the escape character will be ignored.	x in {string}

[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.

6.3. 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
EXTRACT(YEAR|MONTH|DAY|HOUR|MINUTE|SECOND FROM x)	Return the integer value from the date value x. Produces the same result as the assoceated YEAR, MONTH, DAYOFMONTH, HOUR, MINUTE, SECOND functions. The SQL specification also allows for TIMEZONE_HOUR and TIMEZONE_MINUTE as extraction targets. In Teiid all date values are in the timezone of the server.	x in {date, time, 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)
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

Function	Definition	Datatype Constraint
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
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

Function	Definition	Datatype Constraint
TIMESTAMPADD(interval, count, timestamp)	Add a specified interval (hour, day of week, month) to the timestamp, where intervals can have the following definition: 1. SQL_TSI_FRAC_SECOND - fractional seconds (billionths of a second) 2. SQL_TSI_SECOND - seconds 3. SQL_TSI_MINUTE - minutes 4. SQL_TSI_HOUR - hours 5. SQL_TSI_DAY - days 6. SQL_TSI_WEEK - weeks 7. SQL_TSI_MONTH - months 8. SQL_TSI_QUARTER - quarters (3 months) 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 are approximate and may be less accurate over longer time spans.	Interval in {string}; startTime, endTime in {timestamp}, returns a long.
WEEK(x)	Return week in year	x in {date, timestamp}, returns integer

Function	Definition	Datatype Constraint
YEAR(x)	Return four-digit year	x in {date, timestamp}, returns integer

6.3.1. 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')

6.3.2. 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

6.4. 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

Function	Definition
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.

6.5. 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.

6.6. 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 [, z])	Decode column x using string of value pairs y with optional delimiter z and return the decoded column as a string. If a delimiter is not specified , is used. y has the formate SearchDelimResultDelimSearchDelimResult[DelimDefault] Returns Default if specified or x if there are no matches.	all string

Function	Definition	Datatype Constraint
	Warning Deprecated. Use a CASE expression instead.	
DECODEINTEGER(x, y [, z])	Decode column x using string of value pairs y with optional delimiter z and return the decoded column as an integer. If a delimiter is not specified , is used. y has the formate SearchDelimResultDelimSearchDelimResult[DelimDefault] Returns Default if specified or x if there are no matches. **Warning** Deprecated. Use a CASE expression instead.	all string parameters, return integer

Within each function call, you include the following arguments:

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

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

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

When 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.

6.7. Lookup Function

The Lookup function allows you to cache a key value pair table 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.

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

In the lookup table codeTable, find the row where keyColumn has the value keyValue and return the associated returnColumn value or null if no matching key is found. codeTable must be a string literal that is the fully-qualified name of the target table. returnColumn and key Column must also

be string literals of just the relevant column names. The keyValue can be any expression that must match the datatype of the keyColumn. The return datatype matches that of returnColumn.

Example 6.1. Country Code Lookup

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

A ISOCountryCodes table used to translate country name to ISO codes. One column, CountryName, represents a key column. A second column, CountryCode, would represent the ISO code of the country. Hence, a query to this lookup table would provide a CountryName, shown above as 'US', and expect a CountryCode value 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 cache for all queries, in all sessions, that later access this 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.

See the Caching Guide for more on the caching aspects of the lookup function.

Note

- The keyColumn is expected to contain unique values. If the column contains duplicate values, an exception will be thrown.

6.8. System Functions

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

6.8.1. COMMANDPAYLOAD

Retrieve a string from the command payload or null if no command payload was specified. The command payload is set by the `TeiidStatement.setPayload` method on the Teiid JDBC API extensions on a per-query basis.

```
COMMANDPAYLOAD([key])
```

If the key parameter is provided, the command payload object is cast to a java.util.Properties object and the corresponding property value for the key is returned. If the key is not specified the return value is the command payload object toString value.

key, return value are strings

6.8.2. ENV

Retrieve a system environment property.

```
ENV(key)
```

To prevent untrusted access to system properties, this function is not enabled by default. The ENV function may be enabled via the allowEnvFunction property in the file.

key, return value are strings

6.8.3. SESSION_ID

Retrieve the string form of the current session id.

```
SESSION_ID()
```

return value is string.

6.8.4. USER

Retrieve the name of the user executing the query.

```
USER()
```

return value is string.

6.8.5. CURRENT_DATABASE

Retrieve the catalog name of the database. The VDB name is always the catalog name.

```
CURRENT_DATABASE()
```

return value is string.

6.9. XML Functions

XML functions provide functionality for working with XML data.

6.9.1. JSONTOXML

Returns an xml document from JSON.

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

rootElementName is a string, json is in {clob, blob}. Return value is xml.

The appropriate UTF encoding (8, 16LE. 16BE, 32LE, 32BE) will be detected for JSON blobs. If another encoding is used, see the to_chars function.

The result is always a well-formed XML document.

The mapping to XML uses the following rules:

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

Example 6.2. Sample JSON to XML for jsonToXml('person', x)

JSON:

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

XML:

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

Example 6.3. Sample JSON to XML for jsonToXml('person', x) with a root array.

JSON:

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

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

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

6.9.2. XMLCOMMENT

Returns an xml comment.

```
XMLCOMMENT(comment)
```

Comment is a string. Return value is xml.

6.9.3. 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.

6.9.4. 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 6.4. XMLELEMENT of mixed values

with an xml_value of <doc/>,

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

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

6.9.5. 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 [62]*.

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.

6.9.6. 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.

6.9.7. XMLPI

Returns an xml processing instruction.

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

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

6.9.8. 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 [62]*.

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

See also *Section 14.4, "XQuery Optimization"*

6.9.9. 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.

6.9.10. 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.

6.9.11. 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.

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

Example 6.5. Sample xpathValue Ignoring Namespaces

XML value:

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

Function:

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

Results in 'Hello World'

See also *Section 6.9.8, “XMLQUERY”*

6.10. Security Functions

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

6.10.1. HASROLE

Whether the current caller has the Teiid data 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'.

Role names are case-sensitive and only match Teiid *Chapter 10, Data Roles*. JAAS roles/groups names are not valid for this function - unless there is corresponding data role with the same name.

6.11. Miscellaneous Functions

Other functions.

6.11.1. array_get

Retuns the object value at a given array index.

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

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

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

6.11.2. array_length

Returns the length for a given array

```
array_length(array)
```

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

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

6.11.3. uuid

Retuns a universally unique identifier.

```
uuid()
```

the return type is string.

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

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

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

4. Command Deterministic - the result of function evaluation is only deterministic within the scope of the user command. This category include the curdate, curtime, now, and commandpayload functions. Command deterministic functions are delayed in evaluation until processing to ensure that even prepared plans utilizing these functions will be executed with relevant values. Command deterministic function 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.

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

Updatable Views

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

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

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

A UNION ALL can define an inherently updatable view only if each of the UNION branches is itself inherently updatable. A view defined by a UNION ALL can support inherent INSERTs if it is a *Section 14.2.8, "Partitioned Union"* and the INSERT specifies values that belong to a single partition.

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

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

- An INSERT/UPDATE can only modify a single *Section 7.1, "Key-preserved Table"*.
- To allow DELETE operations there must be only a single *Section 7.1, "Key-preserved Table"*.

If the default handling is not available or you wish to have an alternative implementation of an INSERT/UPDATE/DELETE, then you may use *Section 8.3, "Update Procedures"* to define procedures to handle the respective operations.

7.1. Key-preserved Table

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

Procedures

8.1. 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* .

8.1.1. Command Statement

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

Example 8.1. Example Command Statements

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

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

8.1.2. 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 8.2. 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 8.3. 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 8.4. Example Assignment

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

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

Example 8.5. 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 8.6. 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 8.7. 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.

8.1.3. 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.
- The assignment value follows the same rules as for an Assignment Statement.

8.1.4. Assignment Statement

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

Usage:

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

Example Syntax

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

8.1.4.1. Special Variables

`VARIABLES.ROWCOUNT` integer variable will contain the numbers of rows affected by the last insert/update/delete command statement executed. Inserts that are processed by dynamic sql with an into clause will also update the ROWCOUNT.

Example 8.8. Sample Usage

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

8.1.5. Compound Statement

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

Usage:

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

Note

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

Syntax Rules

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

8.1.6. If Statement

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

Usage:

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

Example 8.9. Example If Statement

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

```
END
```

Tip

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.

8.1.7. Loop Statement

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

Usage:

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

Syntax Rules

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

8.1.8. While Statement

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

Usage:

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

Syntax Rules

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

8.1.9. Continue Statement

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

Usage:

```
CONTINUE [label];
```

Syntax Rules

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

8.1.10. Break Statement

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

Usage:

```
BREAK [label];
```

Syntax Rules

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

8.1.11. Leave Statement

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

Usage:

```
LEAVE label;
```

Syntax Rules

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

8.1.12. Error Statement

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

Usage:

```
ERROR message;
```

Example 8.10. Example Error Statement

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

8.2. 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.

8.2.1. Virtual Procedure Definition

Usage:

```
CREATE VIRTUAL PROCEDURE
block
```

The CREATE VIRTUAL PROCEDURE line demarcates the beginning of the procedure. Within the body of the procedure, any valid *statement* may be used.

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

8.2.2. Procedure Parameters

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

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

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

Example 8.11. Example of Referencing an Input Parameter and Assigning an Out Parameter for 'GetBalance' Procedure

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

```
END
```

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

8.2.3. Example Virtual Procedures

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

Example 8.12. 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 8.13. 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
```

8.2.4. 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. Typically you'll use the following syntax:

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

8.2.5. Limitations

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

8.3. 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.

8.3.1. Update Procedure Processing

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

8.3.2. For Each Row

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

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

```
FOR EACH ROW
 BEGIN ATOMIC
 --before row logic

 --default insert/update/delete against the target view
 INSERT INTO VW (c1, c2, c3) VALUES (NEW.c1, NEW.c2, NEW.c3);

 --after row logic
 END
```

8.3.2.1. Definition

Usage:

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

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

Note

The use of the atomic keyword is currently optional for backward compatibility, but unlike a normal block, the default for instead of triggers is atomic.

8.3.2.2. Special Variables

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

8.3.2.2.1. NEW Variables

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

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

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

8.3.2.2.2. OLD Variables

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

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

8.3.2.2.3. CHANGING 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. A CHANGING variable is commonly used to differentiate between a default insert value and one specified in the user query.

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

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

8.3.2.3. Examples

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

Example 8.14. Sample DELETE Procedure

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

Example 8.15. Sample UPDATE Procedure

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

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 9.1. 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.

9.1. 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 9.2. 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.

9.2. 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 9.3. 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.

9.3. JDBC and Transactions

9.3.1. 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.

9.3.2. 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.

9.4. Transactional Behavior with JBoss Data Source Types

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

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

If you have three different sources A, B, C and they are being used in Teiid. Here are some variations on how they behave with different types of data sources. The suffixes "xa", "local", "no-tx" define different type of sources used.

- A-xa B-xa, C-xa : Can participate in all transactional scopes. No restrictions.
- A-xa, B-xa, c-local: Can participate in all transactional scopes. Note that there is only one single source is "local". It is assumed that in the Global scope, the third party datasource, other than Teiid Datasource is also XA.

- A-xa, B-xa, C-no-tx : Can participate in all transactional scopes. Note "C" is not a really bound by any transactional contract. A and B are the only participents in XA transaction.
- A-xa, B-local, C-no-tx : Can participate in all transactional scopes. Note "C" is not a really bound by any transactional contract, and there is only single "local" source.
- If any two or more sources are "local" : They can only participate in Command mode with "autoCommitTxn=OFF". Otherwise will end with exception as "Could not enlist in transaction on entering meta-aware object!;" exception, as it is not possible to do a XA transaction with "local" datasources.
- A-no-tx, B-no-tx, C-no-tx : Can participate in all transaction scopes, but none of the sources will be bound by transactional terms. This is equivalent to not using transactions or setting Command mode with "autoCommitTxn=OFF".

Note

Teiid Designer creates "local" data source by default which is not optimal for the XA transactions. Teiid would like this to be creating a XA data sources, however with current limitations with DTP that feature is currently not available. To create XA data source, look in JBoss AS "doc" directory for example templates, or use the "admin-console" to create the XA data sources.

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

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

Table 9.4. Teiid Transaction Participation

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

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

9.5. 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.

Data Roles

Data roles, also called entitlements, are sets of permissions defined per VDB that dictate data access (create, read, update, delete). Data roles use a fine-grained permission system that Teiid will enforce at runtime and provide audit log entries for access violations (see that Admin and Developers Guide for more on Audit Logging).

Prior to applying data roles, you should consider restricting source system access through the fundamental design of your VDB. Foremost, Teiid can only access source entries that are represented in imported metadata. You should narrow imported metadata to only what is necessary for use by your VDB. When using Teiid Designer, you may then go further and modify the imported metadata at a granular level to remove specific columns, mark tables as non-updatable, etc.

If data roles is enabled and data roles are defined in a VDB, then access permissions will be enforced by the Teiid Server. The use of data roles may be disabled system wide via the `<jboss-install>/standalone/configuration/standalone-teiid.xml` file, by setting the property `useDataRoles` to false in the configuration section of the `RuntimeEngineDeployer`.

Warning

Unlike previous versions of Teiid data roles will only be checked if present in a VDB. A VDB deployed without data roles is open for use by any authenticated user.

10.1. Permissions

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

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

Warning

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note

For backwards compatibility RuntimeEngineDeployer.allowFunctionCallsByDefault located in the `<jboss-install>/standalone/configuration/standalone-teiid.xml` file in the `RuntimeEngineDeployer` section defaults to true. This means that to actually require permissions for functions, you need to set this property to false.

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

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

10.2. Role Mapping

Each Teiid data role can be mapped to any number of container roles or any authenticated user. You may control role membership through whatever system the Teiid security domain login modules are associated with. The kit includes example files for use with the UsersRolesLoginModule - see teiid-security-roles.properties.

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

10.3. 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. The "vdb.xml" file is checked against the schema file `vdb-deployer.xsd`, which can be found in the kit under teiid-docs/schema. This example will show a sample "vdb.xml" file with few simple data roles.

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 permissions to read, write access to TableA, but can not delete.

2. RoleB has no permissions that allow access to TableA

3. RoleC has permissions that only allow read access to TableA.column1

Example 10.1. vdb.xml defining RoleA, RoleB, and RoleC

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

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

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

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

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

    </data-role>

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

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

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

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

```
</vdb>
```

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

The "mapped-role-name" defines the container JAAS roles that are assigned the data role. For assigning roles to your users in the JBoss AS, check out the instructions for the selected Login Module. Check the "Admin Guide" for configuring Login Modules. You may also choose to allow any authenticated user to have a data role by setting the any-authenticated attribute value to true on data-role element.

The "allow-create-temporary-tables" data-role boolean attribute is used to explicitly enable or disable temporary table usage for the role. If it is left unspecified, then the value will be determined by the configuration setting allowCreateTemporaryTablesByDefault located in the `<jboss-install>/standalone/configuration/standalone-teiid.xml` file in the `RuntimeEngineDeployer` section.

10.4. System Functions

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

10.5. Customizing

See the Developer's Guide Custom Security Chapter for details on using an alternative authorization scheme.

System Schema

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

11.1. System Tables

11.1.1. VDB, Schema, and Properties

11.1.1.1. SYSADMIN.VDBResources

This table provides the current VDB contents.

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

11.1.1.2. 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

11.1.1.3. SYS.Schemas

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

Column Name	Type	Description
VDBName	string	VDB name
Name	string	Schema name
IsPhysical	boolean	True if this represents a source
UID	string	Unique ID
OID	integer	Unique ID (see note below)
Description	string	Description
PrimaryMetamodelURI	string	URI for the primary metamodel describing the model used for this schema

11.1.1.4. SYS.Properties

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

Column Name	Type	Description
Name	string	Extension property name
Value	string	Extension property value
UID	string	Key unique ID
OID	integer	Unique ID (see note below)

11.1.2. Table Metadata

11.1.2.1. 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
OID	integer	Unique ID (see note below)
Cardinality	integer	Approximate number of rows in the group
Description	string	Description
IsSystem	boolean	True if in system table

11.1.2.2. SYSADMIN.MatViews

This table supplies information about all the materailized views in the virtual database.

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

Column Name	Type	Description
TargetSchemaName	string	Name of the materialized table schema
TargetName	string	Name of the materialized table
Valid	boolean	True if materialized table is currently valid.
LoadState	boolean	The load state, can be one of NEEDS_LOADING, LOADING, LOADED, FAILED_LOAD
Updated	timestamp	The timestamp of the last full refresh.
Cardinality	integer	The number of rows in the materialized view table.

11.1.2.3. 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
IsCaseSensitive	boolean	Element is case-sensitive
IsSigned	boolean	Element is signed numeric value
IsCurrency	boolean	Element represents monetary value
IsAutoIncremented	boolean	Element is auto-incremented in the source
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
MinRange	string	Minimum value
MaxRange	string	Maximum value
DistinctCount	integer	Distinct value count, -1 can indicate unknown

Column Name	Type	Description
NullCount	integer	Null value count, -1 can indicate unknown
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", "Unsearchable"
Format	string	Format of string value
DefaultValue	string	Default value
JavaClass	string	Java class that will be returned
Precision	integer	Number of digits in numeric value
CharOctetLength	integer	Measure of return value size
Radix	integer	Radix for numeric values
GroupUpperName	string	Upper-case full group name
UpperName	string	Upper-case element name
UID	string	Element unique ID
OID	integer	Unique ID (see note below)
Description	string	Description

11.1.2.4. SYS.Keys

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

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

11.1.2.5. SYS.KeyColumns

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

Column Name	Type	Description
VDBName	string	VDB name

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

11.1.3. Procedure Metadata

11.1.3.1. 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
OID	integer	Unique ID (see note below)
Description	string	Description

11.1.3.2. 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

Column Name	Type	Description
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"

11.1.4. Datatype Metadata

11.1.4.1. 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"
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
OID	integer	Unique ID (see note below)
RuntimeType	string	Teiid runtime data type name
BaseType	string	Base type
Description	string	Description of type

Warning

The OID column is guranteed to be unique/consistent only for given version running instance of a VDB. If a different version of the VDB is deployed, these IDs are not guranteed to be the same or unique across both versions of the VDB. Dynamic VDB OIDs are not cluster safe.

11.2. System Procedures

11.2.1. SYS.getXMLSchemas

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

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

11.2.2. SYSADMIN.logMsg

Log a message to the underlying logging system.

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

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

Example 11.1. Example logMsg

```
CALL SYSADMIN.logMsg(msg=>'some debug', context=>'org.something')
```

This will log the message 'some debug' at the default level DEBUG to the context org.something.

11.2.3. SYSADMIN.isLoggable

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

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

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

Example 11.2. Example isLoggable

```
IF ((CALL SYSADMIN.isLoggable(context=>'org.something'))
BEGIN
  DECLARE STRING msg;
  // logic to build the message ...
  CALL SYSADMIN.logMsg(msg=>msg, context=>'org.something')
END
```

11.2.4. SYSADMIN.refreshMatView

Returns integer RowsUpdated. -1 indicates a load is in progress, otherwise the cardinality of the table is returned. See the Caching Guide for more.

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

11.2.5. SYSADMIN.refreshMatViewRow

Returns integer RowsUpdated. -1 indicates the materialized table is currently invalid. 0 indicates that the specified row did not exist in the live data query or in the materialized table. See the Caching Guide for more.

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

11.2.6. Metadata Procedures

Note

A `MetadataRepository` must be configured to make a non-temporary metadata update persistent. See the Developers Guide Runtime Metadata Updates section for more.

11.2.6.1. SYSADMIN.setTableStats

Set statistics for the given table.

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

11.2.6.2. SYSADMIN.setColumnStats

Set statistics for the given column.

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

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

11.2.6.3. SYSADMIN.setProperty

Set an extension metadata property for the given record. Extension metadata is typically used by *Chapter 13, Translators*.

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

Setting a value to null will remove the property.

Example 11.3. Example Property Set

```
CALL SYSADMIN.setProperty(uid=>(SELECT uid FROM TABLES WHERE name='tab'),
 name=>'some name', value=>'some value')
```

This will set the property 'some name'='some value' on table tab.

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

VDBs

12.1. VDB Definition

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

Example 12.1. Example VDB XML

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

   <!-- VDB properties -->
   <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>
```

12.1.1. 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 `PROFILE/data/teiid` directory

- *query-timeout*

  Sets the default query timeout in milliseconds for queries executed against this VDB. 0 indicates that the server default query timeout should be used. Defaults to 0. Will have no effect if the server default query timeout is set to a lesser value. Note that clients can still set their own timeouts that will be managed on the client side.

12.1.2. 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

A source is a named binding of a translator and connection source to a model.

- *name*

The name of the source to use for this model. This can be any name you like, but will typically be the same as the model name. Having a name different than the model name is only useful in multi-source scenarios. In multi-source, the source names under a given model must be unique. If you have the same source bound to multiple models it may have the same name for each. An exception will be raised if the same source name is used for different sources.

- *translator-name*

  The name or type of the 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.

12.1.3. 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.

12.2. 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.)

12.3. Multi-Source Models and VDB

Multi-source models can be used to quickly access data in multiple sources with homogeneous metadata. When you have multiple instances of data that are using identical schema (horizontal sharding), Teiid can help you aggregate data across all the instances, using "multi-source" models. In this scenario, instead of creating/importing a model for every data source, user needs to define one source model that represents the schema and configure multiple data "sources" underneath it. During runtime, when a query issued against this model, the query engine analyzes the information and gathers the required data from all the sources configured and aggregates the results and provides in a single result set. Since all sources utilize the same physical metadata, this feature is most appropriate for accessing the same source type with multiple instances.

To mark a model as multi-source, the user needs to supply property called supports-multi-source-bindings, in the "vdb.xml" file. Also, the user needs to define multiple sources. Here is code example showing dynamic vdb with single model with multiple sources defined.

```
<vdb name="vdbname" version="1">
   <model visible="true" type="PHYSICAL" name="Customers" path="/Test/Customers.xmi">
      <property name="supports-multi-source-bindings" value="true"/>
      <source name="chicago"
         translator-name="oracle" connection-jndi-name="chicago-customers"/>
      <source name="newyork"
         translator-name="oracle" connection-jndi-name="newyork-customers"/>
      <source name="la"
         translator-name="oracle" connection-jndi-name="la-customers"/>
   </model>
</vdb>
```

In the above example, the VDB defined has single model called `Customers`, that has multiple sources (`chicago`, `newyork`, and `la`) that define different instances of data. Every time a model is

marked as "multi-source", the runtime engine adds a additional column called "SOURCE_NAME" to every table in that model. This column maps to the source's name from the XML. In the above XML code that would be `chicago`, `la`, `newyork`. This allows queries like the following:

```
select * from table where SOURCE_NAME = 'newyork'
update table column=value  where SOURCE_NAME='chicago'
delete from table where column = x and SOURCE_NAME='la'
insert into table (column, SOURCE_NAME) VALUES ('value', 'newyork')
```

Note that when user do not supply the "SOURCE_NAME" in the criteria, the command applies to all the sources. If SOURCE_NAME supplied, the query is executed only aginst the source specified. Another useful feature along with this feature is "partial results" to skip unavailable sources if they are down.

More complex partitioning scenarios, such as heterogeneous sources or list partitioning will require the use of a *Section 14.2.8, "Partitioned Union"*.

Note

Currently the tooling support for managing the multi-source feature is limited, so if you need to use this feature build the VDB as usual in the Teiid Designer and then edit the "vdb.xml" file in the VDB archive using a Text editor to add the additional sources as defined above. You must deploy a separate data source for each source defined in the xml file.

Note

If you would like to use "SOURCE_NAME" in your transformations to control which sources are accessed or updated, you would manually need to add this extra column on your view table in the Designer. This column will not be automatically added on the source table, when you import the medata from source. It is important to understand that a column or IN procedure parameter named source_name in multi-source mode will always be treated as the explicit form of the multi-source source_name column and will no longer be treated as an actual physical column or procedure parameter.

12.3.1. Multi-source SELECTs

A multi-source SELECT may use the source_name column anywhere a column reference is allowed. As a final stage of planning, a source query will be generated against each source and each instance of the source_name column replaced by the appropriate value. If the resulting query still needs executed, it is sent to the source. If the WHERE clause evaluates to always false, then

the query is pruned from the result. All results are then unioned together and returned as the full result.

12.3.2. Multi-source INSERTs

A multi-source INSERT may use the source_name column as an insert target column to specify which source should be targeted by the INSERT. Only a INSERT using the VALUES clause is supported and the source_name column value must be a literal. If the source_name column is not part of the INSERT column, then the INSERT will be issued against every source. The sum of the update counts will be returned as the resultant update count.

12.3.3. Multi-source UPDATEs

A multi-source delete functions just like SELECT, however it is not possible to use the source_name column as a target column in the change set. Any other usage of the source_name column will be the appropriate value for each source. If the WHERE clause evaluates to always false, then no update will be issued to the source. The sum of the update counts will be returned as the resultant update count.

12.3.4. Multi-source DELETEs

A multi-source delete functions just like SELECT. Any usage of the source_name column will be the appropriate value for each source. If the WHERE clause evaluates to always false, then no delete will be issued to the source. The sum of the update counts will be returned as the resultant update count.

12.3.5. Multi-source Stored Procedures

A physical stored procedures requires the manual addition of a string source_name parameter to allow for specifying which source the procedure is executed on. If the source_name parameter is not added or if named parameters are used and the source_name parameter is allowed to default to a null value, then the procedure will be executed on each source and the results unioned together.

It is not possible to execute procedures that required to return IN/OUT, OUT, or RETURN parameters values on more than 1 source at a time.

12.3.6. Additional Concerns

When running under a transaction of in a mode that detects the need for a transaction and multiple updates are performed, an attempt will be made to enlist each source in the same XA transaction.

Translators

13.1. Introduction to the Teiid Connector Architecture

The Teiid Connector Architecture (TCA) 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. Refer to 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.

13.2. 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 Administrator 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 13.1. 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
MaxInCriteriaSize	If in criteria are supported, defines what the maximum number of in entries are per predicate. -1 indicates no limit.	-1
MaxDependentInPredicates	If in criteria are supported, defines what the maximum number of predicates that can be used for a dependent join. Values less than 1 indicate to use only one in predicate per dependent value pushed (which matches the pre-7.4 behavior).	-1

Note

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

There are no base importer settings.

13.2.1. 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 13.2. Execution Properties

Name	Description	Default
Encoding	The encoding that should be used for CLOBs returned by the getTextFiles procedure	The system default encoding
ExceptionIfFileNotFound	Throw an exception in getFiles or getTextFiles if the specified file/directory does not exist.	false

13.2.1.1. Usage

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

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

If the extension path is specified, then it will filter all of the file in the directory referenced by the base path. If the extension pattern is not specified and the path is a directory, then all files in the directory will be returned. Otherwise the single file referenced will be returned. If the path doesn't exist, then no results will be returned if ExceptionIfFileNotFound is false, otherwise an exception will be raised.

Retrieve all files as CLOB(s) with the an optional extension at the given path.

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

All the same files a getFiles will be retrieved, the only difference is that the results will be CLOB values using the encoding execution property as the character set.

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

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

The path should reference a new file location or an existing file to overwrite completely.

13.2.1.2. JCA Resource Adapter

The resource adapter for this translator provided through "File Data Source", Refer to Admin Guide for configuration information.

13.2.2. 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.
- *access* - for use with Microsoft Access 2003 or later.
- *db2* - for use with DB2 8 or later.
- *derby* - for use with Derby 10.1 or later.
- *h2* - for use with H2 version 1.1 or later.
- *hive* - For use with Hive database based on Hadoop. Hive is a data warehousing infrastructure based on the Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing (using the map-reduce programming paradigm) on commodity hardware.

  Hive has limited support for data types as it supports integer varients, boolean, float, double and string. It is does not have native support for time based types, xml or LOBs. These limitations are reflected in the translator capabilities. The view table can use these types, however the tranformation would need to specify the necessary transformations. Note that in those situations, the evaluations will be done in Teiid engine. Another limitation Hive has is, it only supports EQUI join, so using any other joins types on its source tables will result in in-effiecient queries. Currently there is no tooling support for metadata import from Hive in Designer. To write criteria based on partitioned columns, they can be modeled on source table, but do not include in selection columns.
- *hsql* - for use with HSQLDB 1.7 or later.
- *ingres* - for use with Ingres 2006 or later.
- *ingres93* - for use with Ingres 9.3 or later.
- *intersystems-cache* - for use with Intersystems Cache Object database (only relational aspect of it)
- *informix* - for use with any version.

- *metamatrix* - for use with MetaMatrix 5.5.0 or later.
- *modeshape* - for use with Modeshape 2.2.1 or later. The PATH, NAME, LOCALNODENAME, DEPTH, and SCORE functions should be accessed as pseudo-columns, e.g. "nt:base"."jcr:path". Teiid UFDs (prefixed by JCR_) are available for CONTIANS, ISCHILDNODE, ISDESCENDENT, ISSAMENODE, REFERENCE - see the JCRFunctions.xmi. If a selector name is needed in a JCR function, you should use the pseudo-column "jcr:path", e.g. JCR_ISCHILDNODE(foo.jcr_path, 'x/y') would become ISCHILDNODE(foo, 'x/y') in the ModeShape query. An additional pseudo-column "mode:properties" should be imported by setting the ModeShape JDBC connection property teiidsupport=true. The column "mode:properties" should be used by the JCR_REFERENCE and other functions that expect a .* selector name, e.g. JCR_REFERENCE(nt_base.jcr_properties) would become REFERENCE("nt:base".*) in the ModeShape query.
- *mysql*/*mysql5* - for use with MySQL version 4.x and 5 or later respectively.

  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>`. A rownum column can also added to any Oracle physical table to support the rownum pseudo-column. A rownum colum should have a name in source of `rownum`. These rownum columns do not have the same semantics as the Oracle rownum construct so care must be taken in their usage.

  Oracle specific execution properties:

  - *OracleSuppliedDriver* - indicates that the Oracle supplied driver (typically prefixed by ojdbc) is being used. Defaults to true. Set to false when using DataDirect or other Oracle JDBC drivers.
- *postgresql* - for use with 8.0 or later clients and 7.1 or later server.
- *sqlserver* - for use with SQL Server 2000 or later. A SQL Server JDBC driver version 2.0 or later (or compatible e.g. JTDS 1.2 or later) should be used.
- *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 13.3. 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
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
MaxPreparedInsertBatchSize	The max size of a prepared insert batch.	2048

Table 13.4. 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

Name	Description	Default
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	false will override the default and direct Teiid to create source queries using unquoted identifiers.	true
useProcedureSpecificName	true will allow the import of overloaded procedures (which will normally result in a duplicate procedure error) by using the unique procedure specific name as the Teiid name. This option will only work with JDBC 4.0 compatable drivers that report specific names.	false
useCatalogName	true will use any non-null/non-empty catalog name as part of the name in source, e.g. "catalog"."table"."column", and in the Teiid runtime name if useFullSchemaName is true. false will not use the catalog name in either the name in source or the Teiid runtime name. Should be set to false for sources that do not fully support a catalog concept, but return a non-null catalog name in their metadata - such as HSQL.	true

[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"/>
...
```

13.2.2.1. Usage

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

13.2.2.2. Native Queries

Both physical tables and procedures may optionally have native queries associated with them. No validation of the native query is performed, it is simply used in a straight-forward manner to generate the source SQL. For a physical table setting the teiid:native-query extension metadata to the desired query string will have Teiid execute the native query as an inline view in the source query. This feature should only be used against sources that support inline views. For example on a physical table y with nameInSource=x and teiid:native-query=select c from g, the Teiid source query "SELECT c FROM y" would generate the SQL query "SELECT c FROM (select c from g) as x". Note that the column names in the native query must match the nameInSource of the physical table columns for the resulting SQL to be valid.

For physical procedures you may also set the teiid:native-query extension metadata to a desired query string with the added ability to positionally reference IN parameters. A parameter reference has the form $integer, e.g. $1. Note that 1 based indexing is used and that only IN parameters may be referenced. Dollar-sign integer is reserved in physical procedure native queries. To use a $integer directly, it must be escaped with another $, e.g. $$1. By default bind values will be used for parameter values. In some situations you may wish to bind values directly into the resulting SQL. The teiid:non-prepared extension metadata property may be set to false to turn off parameter binding. Note this option should be used with caution as inbound may allow for SQL injection attacks if not properly validated. The native query does not need to call a stored procedure. Any SQL that returns a result set positionally matching the result set expected by the physical stored procedure metadata will work. For example on a stored procedure x with teiid:native-query=select c from g where c1 = $1 and c2 = '$$1', the Teiid source query "CALL x(?)" would generate the SQL

query "select c from g where c1 = ? and c2 = '$1'". Note that ? in this example will be replaced with the actual value bound to parameter 1.

13.2.2.3. JCA Resource Adapter

The resource adapter for this translator provided through data source in JBoss AS, Refer to Admin Guide for "JDBC Data Sources" configuration section.

13.2.3. LDAP Translator

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

Table 13.5. 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
UsePagination	Use a PagedResultsControl to page through large results. This is not supported by all directory servers.	false
ExceptionOnSizeLimitExceeded	Set to true to throw an exception when a SizeLimitExceededException is received and a LIMIT is not properly enforced.	false

There are no import settings for the ldap translator; it also does not provide metadata.

13.2.3.1. Metadata Directives

String columns with a default value of "multivalued-concat" will concatinate all attribute values together in alphabetical order using a ? delimiter. If a multivalued attribute does not have a default value of "multivalued-concat", then any value may be returned.

13.2.3.2. JCA Resource Adapter

The resource adapter for this translator provided through "LDAP Data Source", Refer to Admin Guide for configuration.

13.2.4. 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 13.6. 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 "asynchronously" 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.

13.2.4.1. JCA Resource Adapter

The source connection is required for this translator

13.2.5. 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 13.7. Execution Properties

Name	Description	Default
ModelAuditFeilds	Audit Model Fields	false

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

13.2.5.1. Usage

13.2.5.1.1. 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*.

13.2.5.1.2. 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';
```

13.2.5.1.3. 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;
```

13.2.5.1.4. Selecting Updated Objects

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

```
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.

13.2.5.1.5. Selecting Deleted Objects

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

```
GetDeleted (ObjectName IN string,
 StartDate IN datetime,
 EndDate IN datetime,
 EarliestDateAvailable OUT datetime,
 LatestDateCovered OUT datetime)
returns
 ID string,
 DeletedDate datetime
```

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

13.2.5.1.6. 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.

13.2.5.1.7. 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

13.2.5.2. JCA Resource Adapter

The resource adapter for this translator provided through "Salesforce Data Source", Refer to Admin Guide for configuration.

13.2.6. 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.

Note

Setting the proper binding value on the translator is recommended as it removes the need for callers to pass an explict value. If your service is actually uses SOAP11, but the binding used SOAP12 you will receive execution failures.

Table 13.8. 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
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.

13.2.6.1. Usage

The WS translator exposes low level procedures for accessing web services. See also the ws-weather example in the kit.

13.2.6.1.1. Invoke Procedure

Invoke allows for multiple binding, or protocol modes, including HTTP, SOAP11, and SOAP12.

```
Procedure invoke(binding in STRING, action in STRING, request in XML, endpoint in STRING)
 returns XML
```

The binding may be one of null (to use the default) HTTP, SOAP11, or SOAP12. Action with a SOAP binding indicates the SOAPAction value. Action with a HTTP binding indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for the binding or endpoint will use the default value. The default endpoint is specified in the WS resource adapter configuration. The endpoint URL may be absolute or relative. If it'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.

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

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

13.2.6.1.2. InvokeHTTP Procedure

`invokeHttp` can return the byte contents of an HTTP(S) call.

```
Procedure invokeHttp(action in STRING, request in OBJECT, endpoint in STRING, contentType
 out STRING) returns BLOB
```

Action indicates the HTTP method (GET, POST, etc.), which defaults to POST.

A null value for endpoint will use the default value. The default endpoint is specified in the WS resource adapter configuration. The endpoint URL may be absolute or relative. If it'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.

```
call invokeHttp(action=>'GET')
```

The request can be one of SQLXML, STRING, BLOB, or CLOB. The request will be sent as the POST payload in byte form. For STRING/CLOB values this will default to the UTF-8 encoding. To control the byte encoding, see the *to_bytes [50]* function.

13.2.6.2. JCA Resource Adapter

Theresource adapter for this translator provided through "Web Service Data Source", Refer to Admin Guide for configuration.

13.2.7. OLAP Translator

The OLAP Services translator, known by the type name *olap*, exposes stored procedures for calling analysis sevices backed by a OLAP server using MDX query lanaguage. This translator exposes a stored procedure, invokeMDX, that returns a result set containing tuple array values for a given MDX query. invokeMDX will commonly be used with the *ARRAYTABLE* table function to extract the results.

Since the Cube metadata exposed by the OLAP servers and relational database metadata are so different, there is no single way to map the metadata from one to other. It is best to query OLAP system using its own native MDX language through. MDX queries my be defined statically or built dynamically in Teiid's abstraction layers.

13.2.7.1. Usage

The olap translator exposes one low level procedure for accessing olap services.

13.2.7.1.1. InvokeMDX Procedure

`invokeMdx` returns a resultset of the tuples as array values.

```
Procedure invokeMdx(mdx in STRING) returns resultset (tuple object)
```

The mdx parameter is a MDX query to be executed on the OLAP server.

The results of the query will be returned such that each row on the row axis will be packed into an array value that will first contain each hierarchy member name on the row axis then each measure value from the column axis.

Note

The use of *Chapter 10, Data Roles* should be considered to prevent arbitrary MDX from being submitted to the invokeMDX procedure.

13.2.7.2. JCA Resource Adapter

The resource adapter for this translator provided through data source in JBoss AS, Refer to Admin Guide for "JDBC Data Sources" configuration section. Two sample -ds.xml files provided for accessing OLAP servers in teiid-examples section. One is Mondrian specific, when Mondrian server is deloyed in the same JBoss AS as Teiid (mondrian-ds.xml). To access any other OLAP servers using XMLA interface, the data source for them can be created using them example template olap-xmla-ds.xml

13.2.8. Delegating Translators

You may create a delegating translator by extending the `org.teiid.translator.BaseDelegatingExecutionFactory`. Once your classes are then packaged as a custom translator, you will be able to wire another translator instance into your delegating translator at runtime in order to intercept all of the calls to the delegate. This base class does not provide any functionality on its own, other than delegation.

Table 13.9. Execution Properties

Name	Description	Default
delegateName	Translator instance name to delegate to	

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.

14.1. 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 *SHOWPLAN 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* .

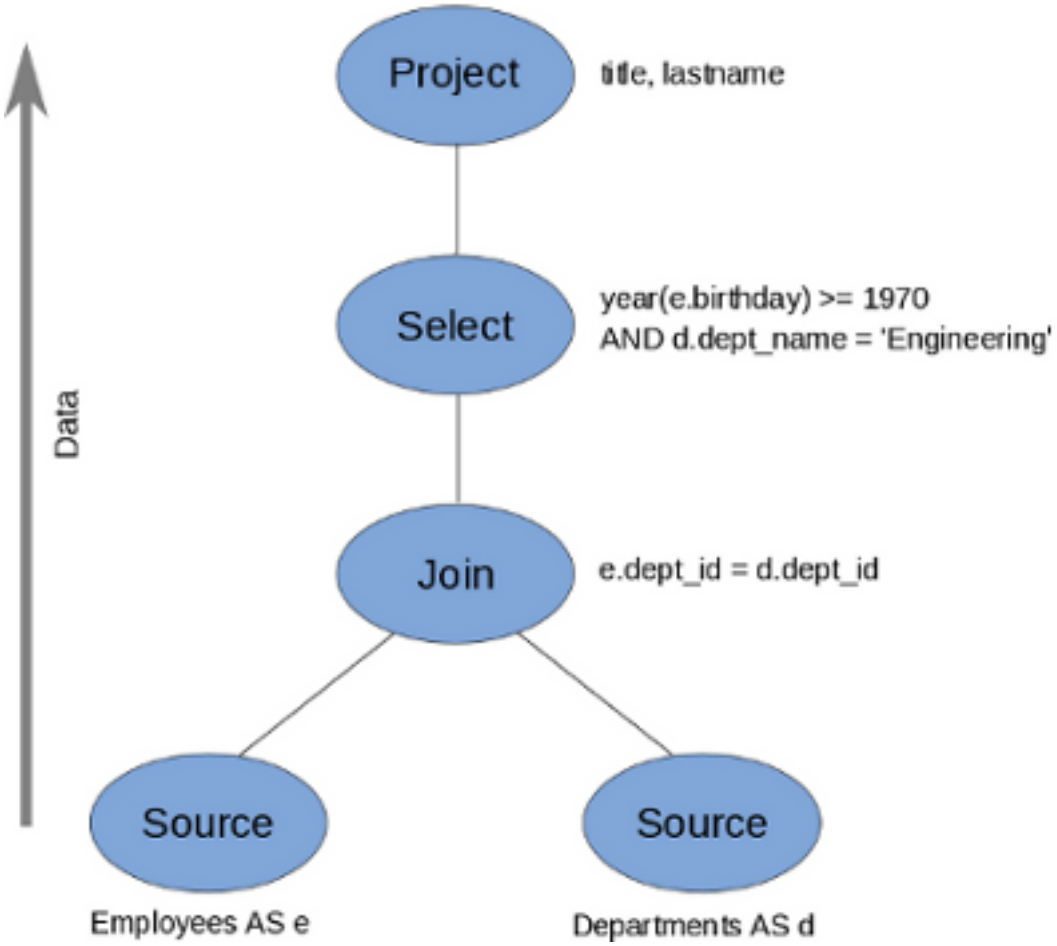
The logical query plan is a tree of operations used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute 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 14.1. Example query

```
SELECT e.title, e.lastname FROM Employees AS e JOIN
Departments AS d ON e.dept_id = d.dept_id WHERE year(e.birthday) >= 1970 AND d.dept_name
 = 'Engineering'
```

Logically, the data from the Employees and Departments tables are retrieved, then joined, then filtered as specified, and finally the output columns are projected. The canonical query plan thus looks like this:



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.

14.2. Federated Optimizations

14.2.1. 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.

14.2.2. 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.

14.2.3. 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 *Section 14.2.1, “Access Patterns”*, hints, and costing information.

Teiid supports hints to control dependent join behavior:

- MAKEIND - indicates that the clause should be the independent side of a depedent join.
- MAKEDEP - indicates that the clause should be the dependent side of a join.
- MAKENOTDEP - prevents the clause from being the dependent side of a join.

Theses can be placed in either the *OPTION clause* or directly in the *FROM clause* . As long as all *Section 14.2.1, "Access Patterns"* can be met, the MAKEIND, MAKEDEP, and MAKENOTDEP hints override any use of costing information. MAKENOTDEP supersedes the other hints.

Tip

The MAKEDEP/MAKEIND 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/MAKEIND hint can force an inefficient join structure and may result in many source queries.

The engine will for IN clauses to filter the values coming from the dependent side. If the number of values from the independent side exceeds the translators MaxInCriteriaSize, the values will be split into multiple IN predicates up to MaxDependentPredicates. When the number of independent values exceeds MaxInCriteriaSize*MaxDependentPredicates, then multiple dependent queries will be issued in parallel.

14.2.4. 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

14.2.5. 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.

14.2.6. 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.

14.2.7. Optional Join

The optional join hint indicates to the optimizer that a joined table should be omitted if none of its columns are used by the output of the user query or in a meaningful way to construct the results of the user query. This hint is typically only used in view layers containing multi-source joins.

The optional join hint is applied as a comment on a join clause. It can be applied in both ANSI and non-ANSI joins. With non-ANSI joins an entire joined table may be marked as optional.

Example 14.2. Example Optional Join Hint

```
select a.column1, b.column2 from a, /*+ optional */ b WHERE a.key = b.key
```

Suppose this example defines a view layer X. If X is queried in such a way as to not need b.column2, then the optional join hint will cause b to be omitted from the query plan. The result would be the same as if X were defined as:

```
select a.column1 from a
```

Example 14.3. Example ANSI Optional Join Hint

```
select a.column1, b.column2, c.column3 from /*+ optional */ (a inner join b ON a.key = b.key)
 INNER JOIN c ON a.key = c.key
```

In this example the ANSI join syntax allows for the join of a and b to be marked as optional. Suppose this example defines a view layer X. Only if both column a.column1 and b.column2 are not needed, e.g. "SELECT column3 FROM X" will the join be removed.

The optional join hint will not remove a bridging table that is still required.

Example 14.4. Example Briding Table

```
select a.column1, b.column2, c.column3 from /*+ optional */ a, b, c WHERE ON a.key = b.key
 AND a.key = c.key
```

Suppose this example defines a view layer X. If b.column2 or c.column3 are solely required by a query to X, then the join on a be removed. However if a.column1 or both b.column2 and c.column3 are needed, then the optional join hint will not take effect.

Tip

When a join clause is omitted via the optional join hint, the relevant criteria is not applied. Thus it is possible that the query results may not have the same cardinality or even the same row values as when the join is fully applied.

Left/right outer joins where the inner side values are not used and whose rows under go a distinct operation will automatically be treated as an optional join and do not require a hint.

Example 14.5. Example Unnecessary Optional Join Hint

```
select a.column1, b.column2 from a LEFT OUTER JOIN /*+optional*/ b ON a.key
= b.key
```

Warning

A simple "SELECT COUNT(*) FROM VIEW" against a view where all join tables are marked as optional will not return a meaningful result.

14.2.8. Partitioned Union

Union partitioning is inferred from the transformation/inline view. If one (or more) of the UNION columns is defined by constants and/or has WHERE clause IN predicates containing only constants that make each branch mutually exclusive, then the UNION is considered partitioned. UNION ALL must be used and the UNION cannot have a LIMIT, WITH, or ORDER BY clause (although individual branches may use LIMIT, WITH, or ORDER BY). Partitioning values should not be null. For example the view definition "select 1 as x, y from foo union all select z, a from foo1 where z in (2, 3)" would be considered partitioned on column x, since the first branch can only be the value 1 and the second branch can only be the values 2 or 3. Note that more advanced or explicit partition could be considered in the future. The concept of a partitioned union is used for performing partition-wise joins, in *Chapter 7, Updatable Views*, and *Section 14.2.6, "Partial Aggregate Pushdown"*.

14.2.9. 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.
- *Section 14.3, "Subquery optimization"*

14.3. Subquery optimization

- EXISTS subqueries are typically rewrite to "SELECT 1 FROM ..." to prevent unnecessary evaluation of SELECT expressions.
- Quantified compare SOME subqueries are always turned into an equivalent IN prediate or comparison against an aggregate value. e.g. col > SOME (select col1 from table) would become col > (select min(col1) from table)
- Uncorrelated EXISTs and scalar subquery that are not pushed to the source can be preevaluated prior to source command formation.
- Correlated subqueries used in DETELEs or UPDATEs that are not pushed as part of the corresponding DELETE/UPDATE will cause Teiid to perform row-by-row compensating processing. This will only happen if the affected table has a primary key. If it does not, then an exception will be thrown.
- WHERE or HAVING clause IN and EXISTs predicates can take the MJ (merge join), DJ (dependent join), or NO_UNNEST (no unnest) hints appearing just before the subquery. The MJ hint directs the optimizer to use a traditional, semijoin, or antisemijoin merge join if possible. The DJ is the same as the MJ hint, but additional directs the optimizer to use the subquery as the independent side of a dependent join if possible. The NO_UNNEST hint, which supercedes the other hints, will direct the optimizer to leave the subquery in place.

Example 14.6. Merge Join Hint Usage

```
SELECT col1 from tbl where col2 IN /*+ MJ */ (SELECT col1 FROM tbl2)
```

Example 14.7. Dependent Join Hint Usage

```
SELECT col1 from tbl where col2 IN /*+ DJ */ (SELECT col1 FROM tbl2)
```

Example 14.8. No Unnest Hint Usage

```
SELECT col1 from tbl where col2 IN /*+ NO_UNNEST */ (SELECT col1 FROM tbl2)
```

- The system property *org.teiid.subqueryUnnestDefault* controls whether the optimizer will by default unnest subqueries. The default is false. If true, then most non-negated WHERE or HAVING clause non-negated EXISTS or IN subquery predicates can be converted to a traditional merge join or as antijoin or semijoin variants.

- WHERE clause EXISTs and IN predicates that can be rewriten to a traditional join with the semantics of the semi-join can preserved if the system property *org.teiid.subqueryUnnestDefault* is set to true or the subquery has a MJ hint.
- EXISTs and scalar subqueries that are not pushed down, and not converted to merge joins, are implicitly limited to 1 and 2 result rows respectively.
- Conversion of subquery predicates to nested loop joins is not yet available.

14.4. XQuery Optimization

A technique known as document projection is used to reduce the memory footprint of the context item document. Document projection loads only the parts of the document needed by the relevant XQuery and path expressions. Since document projection analysis uses all relevant path expressions, even 1 expression that could potentially use many nodes, e.g. //x rather than /a/b/x will cause a larger memory footprint. With the relevant content removed the entire document will still be loaded into memory for processing. Document projection will only be used when there is a context item (unnamed PASSING clause item) passed to XMLTABLE/XMLQUERY. A named variable will not have document projection performed. In some cases the expressions used may be too complex for the optimizer to use document projection. You should check the SHOWPLAN DEBUG full plan output to see if the appropriate optimization has been performed.

With additional restrictions, simple context path expressions allow the processor to evaluate document subtrees independently - without loading the full document in memory. A simple context path expression can be of the form "[/][ns:]root/[ns1:]elem/...", where a namespace prefix or element name can also be the * wild card. As with normal XQuery processing if namespace prefixes are used in the XQuery expression, they should be declared using the XMLNAMESPACES clause.

Example 14.9. Streaming Eligible XMLQUERY

```
XMLQUERY('/*:root/*:child' PASSING doc)
```

Rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Example 14.10. Streaming Ineligible XMLQUERY

```
XMLQUERY('//child' PASSING doc)
```

The use of the descendent axis prevents the streaming optimization, but document projection can still be performed.

When using XMLTABLE, the COLUMN PATH expressions have additional restrictions. They are allowed to reference any part of the element subtree formed by the context expression and they may use any attribute value from their direct parentage. Any path expression where it is possible to reference a non-direct ancestor or sibling of the current context item prevent streaming from being used.

Example 14.11. Streaming Eligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS fullchild XML PATH '.', parent_attr string PATH '../@attr', child_val integer)
```

The context XQuery and the column path expression allow the streaming optimization, rather than loading the entire doc in-memory as a DOM tree, each child element will be independently added to the result.

Example 14.12. Streaming Ineligible XMLTABLE

```
XMLTABLE('/*:root/*:child' PASSING doc COLUMNS sibling_attr string PATH '../other_child/@attr')
```

The reference of an element outside of the child subtree in the sibling_attr path prevents the streaming optimization from being used, but document projection can still be performed.

14.5. Federated Failure Modes

14.5.1. Partial Results

Teiid provides the capability to obtain "partial results" in the event of data source unavailability or failure. This is especially useful when unioning information from multiple sources, or when doing a left outer join, where you are 'appending' columns to a master record but still want the record if the extra information is not available.

A source is considered to be 'unavailable' if the connection factory associated with the source issues an exception in response to a query. The exception will be propagated to the query processor, where it will become a warning on the statement. See the Client Guide for more on Partial Results Mode and SQLWarnings.

14.6. Query Plans

When integrating information using a federated query planner, it is useful to be able to view the query plans that are created, to better understand how information is being accessed and processed, and to troubleshoot problems.

A query plan is a set of instructions created by a query engine for executing a command submitted by a user or application. The purpose of the query plan is to execute the user's query in as efficient a way as possible.

14.6.1. 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 14.13. Retrieving a Query Plan

```
statement.execute("set showplan on");
ResultSet rs = statement.executeQuery("select ...");
TeiidStatement tstatement = statement.unwrap(TeiidStatement.class);
PlanNode queryPlan = tstatement.getPlanDescription();
System.out.println(queryPlan);
```

The query plan is made available automatically in several of Teiid's tools.

14.6.2. 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.

14.6.3. 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.

14.6.3.1. 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

14.6.4. Source Hints

Teiid user and transformation queries can contain a meta source hint that can provide additional information to source queries. The source hint has the form /*+ sh(:'arg') source-name:'arg1' ... */ and is expected to appear after the query keyword, e.g. "SELECT /*+ sh:'general hint' my-oracle:'oracle hint' */. The sh arg is optional and is passed to all source queries via the `ExecutionContext.getGeneralHint` method. See the Developer's Guide for more onThe additional args should have source-name that matches the source name assigned to the translator in the VDB. If the source-name matches the hint value will be supplied via the `ExecutionContext.getSourceHint` method. See the Developer's Guide for more on using an ExecutionContext. Each of the arg values has the form of a string literal - it must be surrounded in single quotes and a single quote can be escaped with another signle quote. Only the Oracle translator does anything with source hints by default. The Oracle translator will use either the source hint or the general hint (in that order) if available to form an Oracle hint enclosed in /*+ ... */. Source hints in views will not be passed to the source if the view is used as a pushdown subquery, is joined, or is in a set operation.

14.7. 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

14.7.1. 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 clauses 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
- HAVING: Select node
- SELECT: Project node and DupRemoval node (for SELECT DISTINCT)
- INTO: Project node with a SOURCE Node
- ORDER BY: Sort 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 *Section 14.2.1, "Access Patterns"*.
- 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 *Section 14.2.1, “Access Patterns”* dependencies are met. This rule has three main steps. First it must determine an ordering of joins that satisfy the access patterns present. Second it will heuristically create joins that can be pushed to the source (if a set of joins are pushed to the source, we will not attempt to create an optimal ordering within that set. More than likely it will be sent to the source in the non-ANSI multi-join syntax and will be optimized by the database). Third it will use costing information to determine the best left-linear ordering of joins performed in the processing engine. This third step will do an exhaustive search for 6 or less join sources and is heuristically driven by join selectivity for 7 or more sources.

- 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.
- RuleDecomposeJoin - this rule perfomrs a partition-wise join optimization on joins of *Section 14.2.8, "Partitioned Union"*. The decision to decompose is based upon detecting that each side of the join is a partitioned union (note that non-ansi joins of more than 2 tables may cause the optimization to not detect the appropriate join). The rule currently only looks for situations where at most 1 partition matches from each side.
- 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.
- RulePushLimit - pushes limit and offset information as far as possible in the plan.

14.7.2. 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.

14.7.3. 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

15.1. 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.
- 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*.

15.2. Data Management

15.2.1. 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.

15.2.2. 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.

15.2.2.1. 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.

15.2.2.2. 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).

15.2.3. 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.

15.3. Query Termination

15.3.1. 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.

15.3.2. User Query Timeouts

User query timeouts in Teiid can be managed on the client-side or the server-side. Timeouts are only relevant for the first record returned. If the first record has not been received by the client within the specified timeout period, a 'cancel' command is issued to the server for the request and

no results are returned to the client. The cancel command is issued asynchronously without the client's intervention.

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

Server-side timeouts start when the query is received by the engine. There may be a skew from the when the client issued the query due to network latency or server load that may slow the processing of IO work. The timeout will be cancelled if the first result is sent back before the timeout has ended. See the *Chapter 12, VDBs* section for more on setting the query-timeout VDB property. See the Admin Guide for more on modifying the file to set default query timeout for all queries.

15.4. Processing

15.4.1. Join Algorithms

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

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

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

Joins involving equi-join predicates are also eligible to be made into *Section 14.2.3, "Dependent Joins"*.

15.4.2. 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"> | <ARRAY_AGG: "array_agg"> | <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"> | <EACH: "each"> | <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"> | <LEAVE: "leave"> | <LIKE: "like"> | <LIKE_REGEX: "like_regex"> | <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"> | <OFFSET: "offset"> | <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"> | <POS_REF: ["$"] (<DIGIT>)+> | <INTEGERVAL: (<MINUS>)? (<DIGIT>)+> | <FLOATVAL: (<MINUS>)? (<DIGIT>)* <PERIOD> (<DIGIT>)+ (["e","E"] (["+","-"])? (<DIGIT>)+)?> | <STRINGVAL: ("N" | "E")? "\'" ("\\'" | ~["\'"])* "\'"> | <#LETTER: ["a"-"z","A"-"Z"] | ["\u0153"-"\ufffd"]> | <#DIGIT: ["0"-"9"]> }

<DEFAULT> TOKEN : { <COMMA: ","> | <PERIOD: "."> | <LPAREN: "("> | <RPAREN: ")"> | <LBRACE: "{"> | <RBRACE: "}"> | <LSBRACE: "["> | <RSBRACE: "]"> | <EQ: "="> | <NE: "<>"> | <NE2: "!="> | <LT: "<"> | <LE: "<="> | <GT: ">"> | <GE: ">="> | <STAR: "*"> | <SLASH: "/"> | <PLUS: "+"> | <MINUS: "-"> | <QMARK: "?"> | <DOLLAR: "$"> | <SEMICOLON: ";"> | <COLON: ":"> | <CONCAT_OP: "||"> }

A.2. NON-TERMINALS

stringVal	::= (<STRINGVAL>)
id	::= (<ID>)
command	::= (*createUpdateProcedure* | *userCommand* | *callableStatement*) (<SEMICOLON>)? <EOF>
designerCommand	::= (*updateProcedure* | *userCommand*) (<SEMICOLON>)? <EOF>
updateProcedure	::= (*createUpdateProcedure* | *forEachRowTriggerAction*) <EOF>
createTrigger	::= <CREATE> <TRIGGER> <ON> *id nonReserved* <OF> (<INSERT> | <UPDATE> | <DELETE>) <AS> *forEachRowTriggerAction*
alter	::= <ALTER> ((*nonReserved id* <AS> *queryExpression*) | (<PROCEDURE> *id* <AS> *statement*) | (<TRIGGER> <ON> *id nonReserved* <OF> (<INSERT> | <UPDATE> | <DELETE>) ((<AS> *forEachRowTriggerAction*) | *nonReserved*)))
forEachRowTriggerAction	::= <FOR> <EACH> <ROW> ((<BEGIN> (<ATOMIC>)? (*statement*)* <END>) | *statement*)
userCommand	::= (*queryExpression* | *storedProcedure* | *insert* | *update* | *delete* | *dropTable* | *createTempTable* | *alter* | *createTrigger*)
dropTable	::= <DROP> <TABLE> *id*
createTempTable	::= <CREATE> <LOCAL> <TEMPORARY> <TABLE> *id* <LPAREN> *tableElement* (<COMMA> *tableElement*)* (<COMMA> <PRIMARY> *nonReserved* <LPAREN> *id* (<COMMA> *id*)* <RPAREN>)? <RPAREN>
tableElement	::= *id* (*dataTypeString* | *nonReserved*) (<NOT> <NULL>)?

errorStatement	::= <ERROR> *expression*
statement	::= (((*id* <COLON>)? (*loopStatement* | *whileStatement* | *compoundStatement*)) | (*ifStatement* | *delimitedStatement*))
delimitedStatement	::= (*assignStatement* | *sqlStatement* | *errorStatement* | *declareStatement* | *branchingStatement*) <SEMICOLON>
compoundStatement	::= <BEGIN> ((<NOT>)? <ATOMIC>)? (*statement*)* <END>
branchingStatement	::= (((<BREAK> | <CONTINUE>) (*id*)?) | (<LEAVE> *id*))
whileStatement	::= <WHILE> <LPAREN> *criteria* <RPAREN> *statement*
loopStatement	::= <LOOP> <ON> <LPAREN> *queryExpression* <RPAREN> <AS> *id* *statement*
ifStatement	::= <IF> <LPAREN> *criteria* <RPAREN> *statement* (<ELSE> *statement*)?
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* ((*nonReserved* | <EQ>) *assignStatementOperand*)?
assignStatement	::= *id* (*nonReserved* | <EQ>) *assignStatementOperand*
assignStatementOperand	::= ((*insert*) | *update* | *delete* | (*expression*) | *queryExpression*)
sqlStatement	::= ((*userCommand*) | *dynamicCommand* | (*id* (*nonReserved* | <EQ>) *storedProcedure*))
translateCriteria	::= <TRANSLATE> *criteriaSelector* (<WITH> <LPAREN> *id* <EQ> *expression* (<COMMA> *id* <EQ> *expression*)* <RPAREN>)?
createUpdateProcedure	::= <CREATE> (<VIRTUAL>)? (<UPDATE>)? <PROCEDURE> *statement*
dynamicCommand	::= (<EXECUTE> | <EXEC>) ((<STRING> | <IMMEDIATE>))? *expression* (<AS> *createElementsWithTypes* (<INTO> *id*)?)? (<USING> *setClauseList*)? (<UPDATE> ((<INTEGERVAL>) | (<STAR>)))?
setClauseList	::= *id* <EQ> (<COMMA> *id* <EQ>)*
createElementsWithTypes	::= *id dataTypeString* (<COMMA> *id dataTypeString*)*
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> (<GT>)? *expression* (<COMMA> *id* <EQ> (<GT>)? *expression*)*)
insert	::= <INSERT> <INTO> *id* (*columnList*)? ((<VALUES> <LPAREN> *expressionList* <RPAREN>) | (*queryExpression*)) (*option*)?
columnList	::= <LPAREN> *id* (<COMMA> *id*)* <RPAREN>
expressionList	::= *expression* (<COMMA> *expression*)*
update	::= <UPDATE> *id* <SET> *setClauseList* (*where*)? (*option*)?
delete	::= <DELETE> <FROM> *id* (*where*)? (*option*)?
queryExpression	::= (<WITH> *withListElement* (<COMMA> *withListElement*)*)? *queryExpressionBody*
withListElement	::= *id* (*columnList*)? <AS> <LPAREN> *queryExpression* <RPAREN>
queryExpressionBody	::= *queryTerm* ((<UNION> | <EXCEPT>) (<ALL> | <DISTINCT>)? *queryTerm*)* (*orderby*)? (*limit*)? (*option*)?
queryTerm	::= *queryPrimary* (<INTERSECT> (<ALL> | <DISTINCT>)? *queryPrimary*)*
queryPrimary	::= (*query* | (<TABLE> *id*) | (<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>
orderedAgg	::= (<XMLAGG> | <ARRAY_AGG>) <LPAREN> *expression* (*orderby*)? <RPAREN> *filterClause*
textAgg	::= *nonReserved* <LPAREN> <FOR> *derivedColumn* (<COMMA> *derivedColumn*)* (<ID> *charVal*)? ((<ID> *charVal*))? (<ID>)? ((<ID> *id*))? (*orderby*)? <RPAREN> *filterClause*
aggregateSymbol	::= (((*nonReserved* <LPAREN> <STAR> <RPAREN>) | (*nonReserved* <LPAREN> <RPAREN>) | ((*nonReserved* | <ANY> | <SOME>) <LPAREN> (<DISTINCT> | <ALL>)? *expression* <RPAREN>)) *filterClause*)
filterClause	::= (<FILTER> <LPAREN> <WHERE> *booleanPrimary* <RPAREN>)?
from	::= <FROM> (*tableReference* (<COMMA> *tableReference*)*)

tableReference	::= ((<LBRACE> *nonReserved* *joinedTable* <RBRACE>) | *joinedTable*)
joinedTable	::= *tablePrimary* ((*crossJoin* | *qualifiedJoin*))*
crossJoin	::= ((<CROSS> | <UNION>) <JOIN> *tablePrimary*)
qualifiedJoin	::= (((<RIGHT> (<OUTER>)?) | (<LEFT> (<OUTER>)?) | (<FULL> (<OUTER>)?) | <INNER>)? <JOIN> *tableReference* <ON> *criteria*)
tablePrimary	::= (*textTable* | *arrayTable* | *xmlTable* | *unaryFromClause* | *subqueryFromClause* | (<LPAREN> *joinedTable* <RPAREN>)) ((<MAKEDEP>) | (<MAKENOTDEP>))?
xmlSerialize	::= <XMLSERIALIZE> <LPAREN> (*nonReserved*)? *expression* (<AS> (<STRING> | <VARCHAR> | <CLOB>))? <RPAREN>
nonReserved	::= <ID>
arrayTable	::= <ID> <LPAREN> *expression* *nonReserved* *createElementsWithTypes* <RPAREN> (<AS>)? *id*
textTable	::= <ID> <LPAREN> *expression* *nonReserved* *textColumn* (<COMMA> *textColumn*)* (<NO> <ROW> *nonReserved*)? (<ID> *charVal*)? ((<ESCAPE> *charVal*) | (<ID> *charVal*))? (<ID> (*intVal*)?)? (<ID> *intVal*)? <RPAREN> (<AS>)? *id*
textColumn	::= *id* *dataType* (<ID> *intVal* (<NO> *nonReserved*)?)?
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* ((<FOR> *nonReserved*) | (*dataType* (<DEFAULT_KEYWORD> *expression*)? (*nonReserved* *stringVal*)?))
intVal	::= <INTEGERVAL>
subqueryFromClause	::= (<TABLE>)? <LPAREN> (*queryExpression* | *storedProcedure*) <RPAREN> (<AS>)? *id*
unaryFromClause	::= (<ID> ((<AS>)? *id*)?)
where	::= <WHERE> *criteria*
criteria	::= *compoundCritOr*
compoundCritOr	::= *compoundCritAnd* (<OR> *compoundCritAnd*)*
compoundCritAnd	::= *notCrit* (<AND> *notCrit*)*
notCrit	::= (<NOT>)? *booleanPrimary*

booleanPrimary	::= (*translateCriteria* | (*commonValueExpression* ((*betweenCrit* | *matchCrit* | *regexMatchCrit* | *setCrit* | *isNullCrit* | *subqueryCompareCriteria* | *compareCrit*))?) | *existsCriteria* | *hasCriteria*)
operator	::= (<EQ> | <NE> | <NE2> | <LT> | <LE> | <GT> | <GE>)
compareCrit	::= *operator commonValueExpression*
subquery	::= <LPAREN> (*queryExpression* | (*storedProcedure*)) <RPAREN>
subqueryAndHint	::= *subquery*
subqueryCompareCriteria	::= *operator* (<ANY> | <SOME> | <ALL>) *subquery*
matchCrit	::= (<NOT>)? (<LIKE> | (<SIMILAR> <TO>)) *commonValueExpression* (<ESCAPE> *charVal* | (<LBRACE> <ESCAPE> *charVal* <RBRACE>))?
regexMatchCrit	::= (<NOT>)? <LIKE_REGEX> *commonValueExpression*
charVal	::= *stringVal*
betweenCrit	::= (<NOT>)? <BETWEEN> *commonValueExpression* <AND> *commonValueExpression*
isNullCrit	::= <IS> (<NOT>)? <NULL>
setCrit	::= (<NOT>)? <IN> ((*subqueryAndHint*) | (<LPAREN> *commonValueExpression* (<COMMA> *commonValueExpression*)* <RPAREN>))
existsCriteria	::= <EXISTS> *subqueryAndHint*
groupBy	::= <GROUP> <BY> *expressionList*
having	::= <HAVING> *criteria*
orderby	::= <ORDER> <BY> *sortSpecification* (<COMMA> *sortSpecification*)*
sortSpecification	::= *sortKey* (<ASC> | <DESC>)? (*nonReserved nonReserved*)?
sortKey	::= *expression*
intParam	::= (*intVal* | <QMARK>)
limit	::= ((<LIMIT> *intParam* (<COMMA> *intParam*)?) | (<OFFSET> *intParam* (<ROW> | <ROWS>) (*fetchLimit*)?) | (*fetchLimit*))
fetchLimit	::= <FETCH> *nonReserved* (*intParam*)? (<ROW> | <ROWS>) <ONLY>
option	::= <OPTION> (<MAKEDEP> *id* (<COMMA> *id*)* | <MAKENOTDEP> *id* (<COMMA> *id*)* | <NOCACHE> (*id* (<COMMA> *id*)*)?)*
expression	::= *criteria*
commonValueExpression	::= (*plusExpression* (<CONCAT_OP> *plusExpression*)*)
plusExpression	::= (*timesExpression* (*plusOperator timesExpression*)*)

plusOperator	::= (<PLUS> | <MINUS>)
timesExpression	::= (*valueExpressionPrimary* (*timesOperator valueExpressionPrimary*)*)
timesOperator	::= (<STAR> | <SLASH>)
valueExpressionPrimary	::= (<QMARK> | <POS_REF> | *literal* | (<LBRACE> *nonReserved function* <RBRACE>) | (*textAgg* (*windowSpecification*)?) | (*aggregateSymbol* (*windowSpecification*)?) | (*aggregateSymbol* (*windowSpecification*)?) | *orderedAgg* (*windowSpecification*)? | (*aggregateSymbol windowSpecification*) | (*function*) | (<ID> (<LSBRACE> *intVal* <RSBRACE>)?) | *subquery* | (<LPAREN> *expression* <RPAREN> (<LSBRACE> *intVal* <RSBRACE>)?) | *searchedCaseExpression* | *caseExpression*)
windowSpecification	::= <OVER> <LPAREN> (<PARTITION> <BY> *expressionList*)? (*orderby*)? <RPAREN>
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> *expression* <FROM> *expression* (<FOR> *expression*)? <RPAREN>) | (*nonReserved* <LPAREN> (<YEAR> | <MONTH> | <DAY> | <HOUR> | <MINUTE> | <SECOND>) <FROM> *expression* <RPAREN>) | (*nonReserved* <LPAREN> ((((<LEADING> | <TRAILING> | <BOTH>) (*expression*)?) | *expression*) <FROM>)? *expression* <RPAREN>) | (*nonReserved* <LPAREN> *expression* <COMMA> *stringConstant* <RPAREN>) | (*nonReserved* <LPAREN> *intervalType* <COMMA> *expression* <COMMA> *expression* <RPAREN>) | *queryString* | ((<LEFT> | <RIGHT> | <CHAR> | <USER> | <YEAR> | <MONTH> | <HOUR> | <MINUTE> | <SECOND> | <XMLCONCAT> | <XMLCOMMENT>) <LPAREN> (*expressionList*)? <RPAREN>) | ((<TRANSLATE> | <INSERT>) <LPAREN> (*expressionList*)? <RPAREN>) | *xmlParse* | *xmlElement* | (<XMLPI> <LPAREN> (<ID> *idExpression* | *idExpression*) (<COMMA> *expression*)? <RPAREN>) | *xmlForest* | *xmlSerialize* | *xmlQuery* | (*id* <LPAREN> (*expressionList*)? <RPAREN>))
stringConstant	::= *stringVal*
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*
dataTypeString	::= (<STRING> | <VARCHAR> | <BOOLEAN> | <BYTE> | <TINYINT> | <SHORT> | <SMALLINT> | <CHAR> | <INTEGER> | <LONG> | <BIGINT> | <BIGINTEGER> | <FLOAT> | <REAL> | <DOUBLE> | <BIGDECIMAL> | <DECIMAL> | <DATE> | <TIME> | <TIMESTAMP> | <OBJECT> | <BLOB> | <CLOB> | <XML>)
dataType	::= *dataTypeString*
intervalType	::= (*nonReserved*)
literal	::= (*stringVal* | <INTEGERVAL> | <FLOATVAL> | <FALSE> | <TRUE> | <UNKNOWN> | <NULL> | ((<BOOLEANTYPE> | <TIMESTAMPTYPE> | <DATETYPE> | <TIMETYPE>) *stringVal* <RBRACE>))