MetaMatrix

1

MetaMatrix Query Support Handbook

5.5.3

Preface

MetaMatrix 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, MetaMatrix simplifies data access for developers and consuming applications.

Commercial development support, production support, and training for MetaMatrix is available through JBoss Inc.

SQL Support

MetaMatrix 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.

MetaMatrix 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 MetaMatrix. See the *grammar* for the exact form of SQL accepted by MetaMatrix.

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 models, 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: <model_name>.<table_spec>
- COLUMN: <model_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.
- Because different data sources organize tables in different ways, some prepending catalog or schema or user information, MetaMatrix allows the 'table_spec' to be a dot-delimited construct.
- Identifiers are not case-sensitive in MetaMatrix.
- The separate parts of an identifier can be quoted, with double quotes. This is not required, but some tools do this automatically. Quotes establish another level of grouping, in addition to the dot delimiters. Quotes should not be used in such a way that the table specification, which may itself have multiple parts, is split between two quoted sections.

Some examples of valid fully-qualified table identifiers are:

- MyModel.MySchema.Portfolios
- "MyModel"."MySchema.Portfolios"

- MyModel.Inventory
- MyModel.MyCatalog.dbo.Authors

Some examples of valid fully-qualified column identifiers are:

- MyModel.MySchema.Portfolios.portfolioID
- "MyModel"."MySchema.Portfolios"."portfolioID"
- MyModel.Inventory.totalPallets
- MyModel.MyCatalog.dbo.Authors.lastName

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

1.2. Expressions

Identifiers, literals, and functions can be combined into expressions. Expressions can be used almost anywhere in a query -- SELECT, FROM (if specifying join criteria, WHERE, GROUP BY, HAVING. However you currently cannot use expressions in an ORDER BY clause.

MetaMatrix supports the following types of expressions:

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

1.2.1. Column Identifiers

Column identifiers are used to specify the output columns in SELECT statements, the columns and their values for INSERT and UPDATE statements, and criteria used in WHERE and FROM clauses. They are also used in GROUP BY, HAVING, and ORDER BY clauses. The syntax for column identifiers was defined in the *Identifiers* section above.

1.2.2. Literals

Literal values represent fixed values. These can any of the 'standard' *data types*.

Syntax Rules:

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

Some examples of simple literal values are:

- `‘abc’`
- `‘isn’’t true’` - use an extra single tick to escape a tick in a string with single ticks
- `5`
- `-37.75e01` - scientific notation
- `100.0` - parsed as double
- `true`
- `false`
- `'\u0027'` - unicode character

1.2.3. Aggregate Functions

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

MetaMatrix supports the following aggregate functions:

- COUNT(*) – count the number of values (including nulls and duplicates) in a group
- COUNT(expression) – count the number of values (excluding nulls) in a group
- SUM(expression) – sum of the values (excluding nulls) in a group
- AVG(expression) – average of the values (excluding nulls) in a group
- MIN(expression) – minimum value in a group (excluding null)
- MAX(expression) – maximum value in a group (excluding null)

Syntax Rules:

- Some aggregate functions may contain a keyword 'DISTINCT' before the expression, indicating that duplicate expression values should be ignored. DISTINCT is not allowed in COUNT(*) and is not meaningful in MIN or MAX (result would be unchanged), so it can be used in COUNT, SUM, and AVG.
- Aggregate functions may only be used in the HAVING or SELECT clauses and may not be nested within another aggregate function.
- Aggregate functions may be nested inside other functions.

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

1.2.4. Case and searched case

MetaMatrix supports two forms of the CASE expression which allows conditional logic in a scalar expression.

Supported forms:

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

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

1.2.5. Scalar subqueries

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

1.2.6. Parameter references

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

1.3. Criteria

Criteria are of two basic forms:

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

Syntax Rules:

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

Some examples of valid criteria are:

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

Comparing null Values

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

1.4. SQL Commands

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

1.4.1. SELECT Command

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

A SELECT command has a number of clauses:

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

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

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

This model can be used to understand many questions about SQL. For example, columns aliased in the SELECT clause can only be referenced by alias in the ORDER BY clause. Without

knowledge of the processing model, this can be somewhat confusing. Seen in light of the model, it is clear that the ORDER BY stage is the only stage occurring after the SELECT stage, which is where the columns are named. Because the WHERE clause is processed before the SELECT, the columns have not yet been named and the aliases are not yet known.

1.4.2. INSERT Command

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

Example Syntax

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

1.4.3. UPDATE Command

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

Example Syntax

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

1.4.4. DELETE Command

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

Example Syntax

- DELETE FROM table [WHERE criteria]

1.4.5. EXECUTE Command

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

Example Syntax

- EXECUTE proc()
- EXECUTE proc(param, ...)
- EXECUTE proc(name1=param1,name4=param4,name7=param7) - named parameter syntax

Named Parameters

The default order of parameter specification is the same as how they are defined in the procedure definition. You can also 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.

1.5. Temp Tables

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

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

Creation syntax:

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

Drop syntax:

- DROP TABLE <temporary table name>

Limitations:

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

- Temp tables do not support update or delete operations.

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

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

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

1.6. SQL Clauses

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

1.6.1. SELECT Clause

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

Usage:

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

Syntax Rules:

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

1.6.2. FROM Clause

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

Example Syntax:

- FROM {table [AS alias]}

- FROM table1 [INNER|LEFT OUTER|RIGHT OUTER|FULL OUTER] JOIN table1 ON join-criteria
- FROM table1 CROSS JOIN table1
- FROM (subquery) [AS alias]
- FROM table1 JOIN table2 MAKEDEP ON join-criteria
- FROM table1 JOIN table2 MAKENOTDEP ON join-criteria
- FROM table1 left outer join */* optional */* table2 ON join-criteria

DEP Hints

MAKEDEP and MAKENOTDEP are hints used to control *dependent join* behavior. They should only be used in situations where the optimizer does not chose the most optimal plan based upon query structure, metadata, and costing information.

1.6.3. WHERE Clause

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

The general form of the WHERE is:

- WHERE *criteria*

1.6.4. GROUP BY Clause

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

The general form of the GROUP BY is:

- GROUP BY expression (,expression)*

Syntax Rules:

- Column references in the group by clause must by to unaliased output columns.
- Expressions used in the group by must appear in the select clause.
- Column references and expessions in the select clause that are not used in the group by clause must appear in aggregate functions.

- If an aggregate function is used in the SELECT clause and no GROUP BY is specified, an implicit GROUP BY will be performed with the entire result set as a single group. In this case, every column in the SELECT must be an aggregate function as no other column value will be fixed across the entire group.
- The group by columns must be of a comparable type.

1.6.5. HAVING Clause

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

Syntax Rules:

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

1.6.6. ORDER BY Clause

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

Usage:

```
ORDER BY column1 [ASC|DESC], ...
```

Syntax Rules:

- Column references in the order by must appear in the select clause.
- 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 MetaMatrix optimizer.

1.6.7. LIMIT Clause

The LIMIT clause specifies a limit on the number of records returned from the SELECT command. An optional offset (the number of rows to skip) can be specified.

Usage:

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

Examples:

- LIMIT 100 - returns the first 100 records (rows 1-100)

- LIMIT 500, 100 - skips 500 records and returns the next 100 records (rows 501-600)

1.6.8. INTO Clause

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.
- MetaMatrix's support for SELECT INTO is similar to MS SQL Server. The target of the INTO clause is a table where the result of the rest select command will be inserted. SELECT INTO should not be used UNION query.

1.6.9. OPTION Clause

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

Usage:

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

Supported options:

- SHOWPLAN - returns the query plan along with the results
- PLANONLY - returns the query plan, but does not execute the command
- MAKEDEP [table, (table)*] - specifies source tables that should be made dependent in the join
- MAKENOTDEP [table, (table)*] - prevents a dependent join from being used
- DEBUG - prints query planner debug information in the log and returns it through the JDBC API

Examples:

- LIMIT 100 - returns the first 100 records (rows 1-100)
- LIMIT 500, 100 - skips 500 records and returns the next 100 records (rows 501-600)

1.7. Set Operations

MetaMatrix support the UNION and UNION ALL set operation as a way of combining the results of SELECT commands

Usage:

```
command UNION [ALL] command [ORDER BY...]
```

Syntax Rules:

- The output columns will be named by the output columns of the first union 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 is specified without all, then the output columns must be comparable types.

1.8. Subqueries

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

Supported subquery types:

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

Supported subquery locations:

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

1.8.1. Inline views

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

Example 1.1. Example Subquery in FROM Clause (Inline View)

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

1.8.2. Subqueries in the WHERE and HAVING clauses

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

Example 1.2. Example Subquery in WHERE Using EXISTS

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

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

Example 1.3. Example Subqueries in WHERE Clause

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

Datatypes

2.1. Supported Types

MetaMatrix 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 2.1. MetaMatrix Runtime Types

Type	Description	Java Runtime Class	JDBC Type	ODBC Type
string	variable length character string with a maximum length of 4000	java.lang.String	VARCHAR	VARCHAR
char	a single Unicode character	java.lang.Character	CHAR	CHAR
boolean	a single bit, or Boolean, with two possible values	java.lang.Boolean	BIT	SMALLINT
byte	numeric, integral type, signed 8-bit	java.lang.Byte	TINYINT	SMALLINT
short	numeric, integral type, signed 16-bit	java.lang.Short	SMALLINT	SMALLINT
integer	numeric, integral type, signed 32-bit	java.lang.Integer	INTEGER	INTEGER
long	numeric, integral type, signed 64-bit	java.lang.Long	BIGINT	NUMERIC
biginteger	numeric, integral type, arbitrary precision	java.lang.BigInteger	NUMERIC	NUMERIC
float	numeric, floating point type, 32-bit IEEE 754 floating-point numbers	java.lang.Float	REAL	FLOAT
double	numeric, floating point type, 64-bit IEEE 754 floating-point numbers	java.lang.String	DOUBLE	DOUBLE
bigdecimal	numeric, floating point type, arbitrary precision	java.math.BigDecimal	NUMERIC	NUMERIC
date	datetime, representing a single day (year, month, day)	java.sql.Date	DATE	DATE

Type	Description	Java Runtime Class	JDBC Type	ODBC Type
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 com.metamatrix.common.types.BlobType

[b] The concrete type is expected to be com.metamatrix.common.types.ClobType

[c] The concrete type is expected to be com.metamatrix.common.types.XMLType

2.2. 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 MetaMatrix detects that an explicit conversion can not be applied implicitly in criteria, the criteria will be treated as false. For example:

```
SELECT * FROM my.group 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.

Table 2.2. Type Conversions

Source Type	Valid Implicit Target Types	Valid Explicit Target Types
string	clob	char, boolean, byte, short, integer, long, biginteger, float, double, bigdecimal, xml
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, float, double, bigdecimal	boolean, byte, short
long	string, biginteger, float, double, bigdecimal	boolean, byte, short, integer
biginteger	string, long, float, bigdecimal	boolean, byte, short, integer 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

2.3. Special Conversion Cases

2.3.1. Conversion of String Literals

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

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

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

2.3.2. Converting to Boolean

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

Type	Literal Value	Boolean Value
String	'true'	true
	'false'	false
	other	false
Numeric	1	true
	0	false
	other	*error*

2.3.3. Date/Time/Timestamp Type Conversions

MetaMatrix 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.fffffffff [a]	TIMESTAMP

[a] fractional seconds are optional

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

2.4. 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 2.3. Escaped Literal Syntax

Datatype	Escaped Syntax
BOOLEAN	{b'true'} or {b'false'}
DATE	{d'yyyy-mm-dd'}
TIME	{t'hh-mm-ss'}
TIMESTAMP	{ts'yyyy-mm-dd hh:mm:ss.fffffffff'} [a]

[a] fractional seconds are optional

Scalar Functions

MetaMatrix provides an extensive set of built-in scalar functions. See also *SQL Support* and *Datatypes* . In addition, MetaMatrix provides the capability for *user defined functions or UDFs* .

3.1. 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
ABS(x)	Absolute value of x	See standard numeric operators above
ACOS(x)	Arc cosine of x	x in {double}, return type is double
ASIN(x)	Arc sine of x	x in {double}, return type is double
ATAN(x)	Arc tangent of x	x in {double}, return type is double
ATAN2(x,y)	Arc tangent of x and y	x, y in {double}, 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}, return type is double
COT(x)	Cotangent of x	x in {double}, return type is double
DEGREES(x)	Convert x degrees to radians	x in {double}, 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
FORMATBIGINTEGER(x, y)	Formats x using format y	

Function	Definition	Datatype Constraint
		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}, 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 {integer, long, float, double, biginteger}, if x is biginteger then return type is biginteger, else double
RADIANS(x)	Convert x radians to degrees	x in {double}, 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 right of the decimal point	x in {integer, float, double} 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}, return type is double
SQRT(x)	Square root of x	x in {double, float}, return type is double
TAN(x)	Tangent of x	x in {double}, return type is double

3.1.1. Parsing Numeric Datatypes from Strings

MetaMatrix 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/j2se/1.4.2/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'		1234567	integer

Input String	Function Call to Format String	Output Value	Output Datatype
	parseInteger(total, '#,##0;-#,##0')		

3.1.2. Formatting Numeric Datatypes as Strings

MetaMatrix 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/j2se/1.4.2/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'

3.2. 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 character x	return type is integer
CHR(x) CHAR(x)	Provide the character for ASCII value x	x in {integer}

Function	Definition	Datatype Constraint
CONCAT(x, y)	Concatenates x and y with ANSI semantics. If x and/or y is null, returns null.	x, y, is 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, is string
INITCAP(x)	Make first letter of each word in string x capital and all others lowercase	x in {string}
INSERT(str1, start, length, str2)	Insert string2 into string1	str1 in {string}, start in {integer}, length in {integer}, str2 in {string}
LCASE(x)	Lowercase of x	x in {string}
LEFT(x, y)	Get left y characters of x	x in {string}, y in {string}, return string
LENGTH(x)	Length of x	return type is integer
LOCATE(x, y)	Find position of x in y starting at beginning of y	x in {string}, y in {string}, return integer
LOCATE(x, y, z)	Find position of x in y starting at z	x in {string}, y in {string}, z in {integer}, return integer
LPAD(x, y)	Pad input string x with spaces on the left to the length of y	x in {string}, y in {integer}, return string
LPAD(x, y, z)	Pad input string x on the left to the length of y using character z	x in {string}, y in {string}, z in {character}, return string
LTRIM(x)	Left trim x of white space	x in {string}, return string
REPEAT(str1,instances)	Repeat string1 a specified number of times	str1 in {string}, instances in {integer} return string
REPLACE(x, y, z)	Replace all y in x with z	x,y,z in {string}, return string
RIGHT(x, y)	Get right y characters of x	x in {string}, y in {string}, return string
RPAD(input string x, pad length y)	Pad input string x with spaces on the right to the length of y	x in {string}, y in {integer}, return string
RPAD(x, y, z)	Pad input string x on the right to the length of y using character z	x in {string}, y in {string}, z in {character}, return string

Function	Definition	Datatype Constraint
RTRIM(x)	Right trim x of white space	x is string, return string
SUBSTRING(x, y)	Get substring from x, from position y to the end of x	y in {integer}
SUBSTRING(x, y, z)	Get substring from x from position y with length z	y, z in {integer}
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}
UCASE(x)	Uppercase of x	x in {string}

3.3. Date/Time Functions

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

Function	Definition	Datatype Constraint
CURDATE()	Return current date	returns date
CURTIME()	Return current time	returns time
NOW()	Return current timestamp (date and time)	returns timestamp
DAYNAME(x)	Return name of day	x in {date, timestamp}, returns string
DAYOFMONTH(x)	Return day of month	x in {date, timestamp}, returns integer
DAYOFWEEK(x)	Return day of week (Sunday=1)	x in {date, timestamp}, returns integer
DAYOFYEAR(x)	Return Julian day number	x in {date, timestamp}, returns integer
FORMATDATE(x, y)	Format date x using format y	x is date, y is string, returns string
FORMATTIME(x, y)	Format time x using format y	x is time, y is string, returns string
FORMATTIMESTAMP(x, y)	Format timestamp x using format y	x is timestamp, y is string, returns string
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
MONTH(x)	Return month	

Function	Definition	Datatype Constraint
		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
SECOND(x)	Return seconds	x in {time, timestamp}, returns integer
TIMESTAMPCREATE(date, time)	Create a timestamp from a date and time	date in {date}, time in {time}, returns timestamp
TIMESTAMPADD(interval, count, timestamp)	Add a specified interval (hour, day of week, month) to the timestamp, where intervals can have the following definition: 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}

Function	Definition	Datatype Constraint
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 {date, time, timestamp}, returns a long.
WEEK(x)	Return week in year	x in {date, timestamp}, returns integer
YEAR(x)	Return four-digit year	x in {date, timestamp}, returns integer
MODIFYTIMEZONE (timestamp, startTimeZone, endTimeZone)	Returns a timestamp based upon the incoming timestamp adjusted for the differential between the start and end time zones. i.e. if the server is in GMT-6, then modifytimezone({ts '2006-01-10 04:00:00.0'},'GMT-7', 'GMT-8') will return the timestamp {ts '2006-01-10 05:00:00.0'} as read in GMT-6. The value has been adjusted 1 hour ahead to compensate for the difference between GMT-7 and GMT-8.	startTimeZone and endTimeZone are strings, returns a timestamp
MODIFYTIMEZONE (timestamp, endTimeZone)	Return a timestamp in the same manner as modifytimezone(timestamp, startTimeZone, endTimeZone), but will assume that the startTimeZone is the same as the server process.	Timestamp is a timestamp; endTimeZone is a string, returns a timestamp

3.3.1. Parsing Date Datatypes from Strings

MetaMatrix 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')

3.3.2. 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

3.4. Type Conversion Functions

Within your queries, you can convert between datatypes using the CONVERT or CAST keyword. See also *Data Type Conversions* [http://devcentral.metamatrix.com/tech/expl/DataTypeConversions] .

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

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

3.5. 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
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

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

3.6. Decode Functions

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

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

Within each function call, you include the following arguments:

1. x is the input value for the decode operation. This will generally be a column name.

2. y is the literal string that contains a delimited set of input values and output values.

3. z is an optional parameter on these methods that allows you to specify what delimiter the string specified in y uses.

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

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

When the MetaMatrix 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 MetaMatrix 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 MetaMatrix 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 MetaMatrix Server found in that column.

3.7. Lookup Function

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

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

When you call this function for the first time, the MetaMatrix System caches the lookup group when you first request a value for any combination of codeGroup, returnElement, and keyElement. The MetaMatrix System uses this cached map for all queries, in all sessions, that later access this lookup group. The codeGroup requires use of the fully-qualified name, and the returnElement and keyElement parameters should use shortened column names.

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

Note

- The keyElement column is expected to contain unique key values. If the column contains duplicate values, only the last loaded value will be used for lookup purposes. In some cases, this may cause unexpected results, so it is strongly recommended that only columns without duplicate values be used as the keyElement. The lookup caches can be flushed via the svcmgr.
- Cached lookup groups might consume significant memory. You can limit the number and maximum size of these code groups by setting properties of the QueryService through the MetaMatrix Console.

Function	Definition	Datatype Constraint
LOOKUP(codeGroup, returnElement, keyElement, keyValue)	In the lookup group codeGroup, find the row where keyElement has the	codeGroup must be a fully-qualified string literal containing metadata

Function	Definition	Datatype Constraint
	value keyValue and return the associated returnElement	identifiers, keyValue datatype must match datatype of the keyElement, return datatype matches that of returnElement. returnElement and keyElement parameters should use their shortened names.

3.7.1. Clearing the Cache

You can force a cache clearing by using the expert mode of the svcmgr command, found under the \bin directory of your MetaMatrix server installation.

Launch the appropriate command:

1. `svcmgr.cmd` (Windows)
2. `svcmgr.sh` (Solaris or Linux)

From the command line enter ClearCodeTableCaches.

3.8. System Functions

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

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

Function	Definition	Datatype Constraint
	object and look up the specified key in the object	

3.9. XML Functions

XML functions provide functionality for working with XML data.

Function	Definition	Datatype Constraint
`XPATHVALUE(doc, xpath)`	Takes a document and an XPATH query and returns a string value for the result. An attempt is made to provide a meaningful result for non-text nodes.	Doc and xpath are strings. Return value is a string.

3.10. Security Functions

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

Function	Definition	Datatype Constraint
`hasRole(roleType, roleName)`	Whether the current caller has the role roleName of roleType.	roleType must be one of ('data','admin' , 'repository') and roleName must be a string, the return type is Boolean.

3.11. User Defined Functions

If you need to extends MetaMatrix's scalar function library, then MetaMatrix provides a means to define custom scalar functions or User Defined Functions(UDF). The following steps need to be taken in creating a UDF.

3.11.1. UDF Definition

The FunctionDefinition.xmi file provides metadata to the query engine on User Defined Functions. See our product document on "Creating User-defined Functions" for a more extensive reference on creating that file through the Designer Tool.

The following are used to define a UDF.

- *Function Name* When you create the function name, keep these requirements in mind:
  - You cannot use a reserved word, which includes existing MetaMatrix System function names. You cannot overload existing MetaMatrix System functions.

  - The function name must be unique among user-defined functions for the number of arguments. You can use the same function name for different numbers of types of arguments. Hence, you can overload your user-defined functions.
  - The function name can only contain letters, numbers, and the underscore (_). Your function name must start with a letter.
  - The function name cannot exceed 128 characters.
- *Input Parameters* - defines a type specific signature list. All arguments are considered required.
- *Return Type* - the expected type of the returned scalar value.
- *Pushdown* - can be one of REQUIRED, NEVER, ALLOWED. Indicates the expected pushdown behavior. If NEVER or ALLOWED are specified then a Java implementation of the function should be supplied.
- *invocationClass/invocationMethod* - optional properties indicating the static method to invoke when the UDF is not pushed down.
- *Deterministic* - if the method will always return the same result for the same input parameters.

3.11.2. Source Supported UDF

While MetaMatrix provides an extensive scalar function library, it contains only those functions that can be evaluated within the query engine. In many circumstances, especially for performance, a user defined function allows for calling a source specific function.

For example, suppose you want to use the Oracle-specific functions score and contains:

```
SELECT score(1), ID, FREEDATA FROM Docs WHERE contains(freedata, 'nick', 1) > 0
```

The `score` and `contains` functions are not part of built-in scalar function library. While you could write your own custom scalar function to mimic their behavior, it's more likely that you would want to use the actual Oracle functions that are provided by Oracle when using the Oracle Free Text functionality.

In addition to the normal steps outlined in the section to create and install a function model (FunctionDefinitions.xmi), you will need to extend the appropriate connector(s).

For example, to extend the Oracle Connector

- *Required* - extend OracleCapabilities and set up SCORE and CONTAINS as supported functions (this lets MetaMatrix know that the connector can accept these functions).
- Optionally extend the OracleSQLTranslator to insert new FunctionModifiers to handle translation of these functions. Given that the syntax of these functions is same as other typical functions, this probably isn't needed - the default translation should work.

- Create a new connector type - the easiest way is to export the Oracle ANSI connector type from the Console and just modify the properties such as the connector name (to differentiate it from base Oracle connector) and the capabilities class (to use the extended version) and possibly the translation class (if that was extended for b. Also, connector classpath needs to be extended to include a new jar of your changes above.
- Install the code as an extension module and add your new connector type in the Console.

3.11.3. Non-pushdown Support for User-Defined Functions

Non-pushdown support requires a Java function that matches the metadata supplied in the FunctionDefinitions.xmi file. You must create a Java method that contains the function's logic. This Java method should accept the necessary arguments, which the MetaMatrix System will pass to it at runtime, and function should return the calculated or altered value.

3.11.3.1. Java Code

Code Requirements

- The java class containing the function method must be defined public.
- The function method must be public and static.
- Number of input arguments must match the function metadata defined in section *Install user-defined functions*
- All input arguments defined on function must be java.lang.Object.
- Returned value must be declared as java.lang.Object.
- Any exception can be thrown, but MetaMatrix will rethrow the exception as a `FunctionExecutionException`.

Example 3.1. Sample code

```
package userdefinedfunctions;

public class TempConv {

   /**
    * Converts the given Celsius temperature to Fahrenheit, and returns the
    * value.
    * @param doubleCelsiusTemp
    * @return Fahrenheit
    */
   public static Object celsiusToFahrenheit(Object doubleCelsiusTemp){
```

```
        double celsiusTemp = ((Double)doubleCelsiusTemp).doubleValue();
        double fahrenheitTemp = (celsiusTemp)*9/5 + 32;
        return new Double(fahrenheitTemp);
    }
}
```

3.11.3.2. Post Code Activities

1. After coding the functions you should compile the Java code into a Java Archive (JAR) file, so that you can add it to the MetaMatrix System as an Extension Module.
2. After adding the jar file as an extension module, the name of jar file need to be added to user defined functions classpath using Console tool.

3.11.4. Installing user-defined functions

Once a user-defined function model (FunctionDefinitions.xmi) has been created in in the Designer Tool, it should be installed by replacing the existing version under the Extension Modules (for the Enterprise product this will be done through the Console). That will allow the query engine to know about and use functions

Procedures

4.1. Procedure Language

MetaMatrix 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 virtual tables; these are known as *update procedures* .

4.1.1. Command Statement

A command statement executes a *SQL command* , such as SELECT, INSERT, UPDATE, DELETE, or EXECUTE, against one or more other models (and their underlying data sources).

Example 4.1. Example Command Statements

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

4.1.2. 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. At runtime with the "INTO" clause specified, the dynamic command will actually execute a statement that behaves

like a SELECT INTO. 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 special element symbols 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 "USING.". 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 need for a transaction. Accepted values are (0,1,*). 0 means this is a read-only command, 1 specifies that an update may be performed but will be tied to the success or failure of the dynamic SQL, * specifies that the command may performs 1 or more updates that are not bound to the success or failure of the command and that a transaction is required. 0 is the default value if the clause is not specified.

Example 4.2. 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 = USING.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 4.3. 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 = using.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 = using.LastName)';
 ELSE
   crit = '(Customer.Accounts.Last LIKE using.LastName)';
 IF (AccountAccess.GetAccounts.bday IS NOT NULL)
   crit = '(' || crit || ' and (Customer.Accounts.Birthdate = using.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 4.4. Example Assignment

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

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

Example 4.5. Example Dangerous NULL handling

```
...
criteria = '(' || criteria || ' and (Customer.Accounts.Birthdate = using.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 4.6. Example NULL handling

```
...
criteria = '(' || nvl(criteria, '(1 = 1)') || ' and (Customer.Accounts.Birthdate = using.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 4.7. 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.

4.1.3. 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.

4.1.4. Assignment Statement

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

Usage:

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

Example Syntax

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

4.1.5. If Statement

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

Example 4.8. Example If Statement

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

Note

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

4.1.6. Loop Statement

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

Usage:

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

4.1.7. While Statement

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

Usage:

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

4.1.8. Continue Statement

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

4.1.9. Break Statement

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

4.1.10. Error Statement

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

Example 4.9. Example Error Statement

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

4.2. Virtual Procedures

Virtual procedures are defined using the MetaMatrix 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.

4.2.1. Virtual Procedure Definition

Usage:

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

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

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

4.2.2. Procedure Input Parameters

Virtual procedures can take zero or more input parameters. Each input has the following information that is used during runtime processing:

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

You reference an input to a virtual procedure by using the fully-qualified name of the param (or less if unambiguous). For example, MyModel.MyProc.Param1.

Example 4.10. Example of Referencing an Input Parameter for 'GetBalance' Procedure

```
CREATE VIRTUAL PROCEDURE
BEGIN
```

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

4.2.3. Examle Virtual Procedures

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

Example 4.11. Virtual Procedure Using LOOP, CONTINUE, BREAK

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

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

Example 4.12. Virtual Procedure with Conditional SELECT

```
CREATE VIRTUAL PROCEDURE
BEGIN
  DECLARE string VARIABLES.SORTDIRECTION;
```

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

4.2.4. 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 elements or variables in the procedure.

A virtual procedure call will return a result set just like any SELECT, so you can use this in many places you can use a SELECT. However, within a virtual procedure itself you cannot always use an EXEC directly. Instead, you use the following syntax:

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

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

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

4.3. Update Procedures

Virtual tables are abstractions above physical sources. They typically union or join information from multiple tables, often from multiple data sources or other views. MetaMatrix can perform update operations against virtual tables. Update commands - INSERT, UPDATE, or DELETE - against a virtual table require logic to define how the tables and views integrated by the virtual

table are affected by each type of command. This transformation logic is invoked when an update command is issued against a virtual table. Update procedures define the logic for how a user's update command against a virtual table 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.

4.3.1. Update Procedure Definition

Usage:

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

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

4.3.2. Special Variables

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

4.3.2.1. INPUT Variables

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

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

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

4.3.2.2. CHANGING Variables

Similar to INPUT Variables, every attribute in the virtual table 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 virtual table, these variables are initialized to `true` or `false` depending on whether the INPUT variable was set by the command.

For example, for a virtual table 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

4.3.2.3. ROWS_UPDATED Variable

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

4.3.3. Update Procedure Command Criteria

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

4.3.3.1. HAS CRITERIA

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

Usage:

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

Syntax Rules

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

Some samples of the HAS CRITERIA clause:

SQL	Result
`HAS CRITERIA`	Checks simply whether there was any criteria at all.

SQL	Result
`HAS CRITERIA ON (element1, element2)`	Checks whether the criteria uses element1 and element2.
`HAS = CRITERIA ON (element1)`	Checks whether the criteria has a comparison criteria with = operator.
`HAS LIKE CRITERIA`	Checks whether the criteria has a match criteria using LIKE.

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

4.3.3.2. TRANSLATE CRITERIA

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

Usage:

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

Syntax Rules

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

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

Some samples of the HAS TRANSLATE clause:

SQL	Result
`TRANSLATE CRITERIA`	Translates any user criteria using the default mappings.
TRANSLATE CRITERIA WITH (element1 = 'A', element2 = INPUT.element2 + 2)	Translates any criteria with some additional mappings: element1 is always mapped to 'A' and element2 is mapped to the incoming element2 value + 2.

SQL	Result
`TRANSLATE = CRITERIA ON (element1)`	Translates only criteria that have = comparison operator and involve element1.

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

4.3.4. Update Procedure Processing

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

Transaction Support

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

Table 5.1. MetaMatrix Transaction Scopes

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

5.1. AutoWrap Execution Property

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

Table 5.2. AutoWrap 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 be burdonsome on performance.
OPTIMISTIC	*This is the default setting.* Will not automatically wrap a command in a transaction, instead throw an exception if the command executed is potentially unsafe to execute outside of a transaction.
PESSIMITIC	Will automatically wrap commands in a transaction, but only if the command seems to be unsafe.

The concept of command safety with respect to a transaction is determined by MetaMatrix based upon command type and available metadata. Whenever any INSERT, UPDATE, DELETE, or

EXECUTE (with update count greater than 0) command is detected and the success or failure of that command is not the same as the user level command, then the command is deemed unsafe without a transaction.

The update count may be set on dynamic SQL as part of the command and on all other procedures as part of the procedure metadata in the model.

5.2. JDBC and Transactions

5.2.1. 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 MetaMatrix is consumed through its XADataSource, `com.metamatrix.jdbc.MMDataSource`. JEE containers or data access APIs typically control XA transactions on behalf of application code.

5.2.2. 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.

5.3. Limitations and Workarounds

- The client setting of transaction isolation level is not used. 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.

- Since the client transaction isolation level is not used, MetaMatrix internally assumes a level of READ_COMMITTED. This implies that explicit transactions are not required for user level commands performing multiple reads.
- Temporary tables are not transactional. For example, a global temporary table will retain all inserts performed during a local transaction that was rolled back.
- Connectors may be set to immutable to prevent their participation in transactions. This is useful in situations where update commands are being issued against a source that lacks XA transaction capabilities.

System Tables

6.1. VDB and Model Metadata

6.1.1. System.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

6.1.2. System.Models

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

Column Name	Type	Description
Name	string	Model name
Version	string	Model version
IsPhysical	boolean	True if source model, false for view
SupportsWhereAll	boolean	Model supports queries with no criteria
SupportsOrderBy	boolean	Model supports ORDER BY queries
SupportsJoin	boolean	Model supports queries with joins
SupportsDistinct	boolean	Model supports SELECT DISTINCT queries
SupportsOuterJoin	boolean	Model supports queries with outer joins
MaxSetSize	integer	Max number of values to pass in an IN value set for a dependent join
UID	string	Unique ID
Description	string	Description
PrimaryMetamodelURI	string	

Column Name	Type	Description
		URI for the primary metamodel describing this model

6.1.3. System.ModelProperties

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

Column Name	Type	Description
ModelName	string	Model name
Name	string	Property name
Value	string	Property value
UID	string	Model unique ID

6.2. Table Metadata

6.2.1. System.Groups

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

Column Name	Type	Description
ModelName	string	Model name
FullName	string	Full group 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 model
UpperName	string	Upper-case full group name for easier matching
SupportsUpdates	boolean	True if group can be updated
UID	string	Group unique ID
Cardinality	integer	Approximate number of rows in the group
Description	string	Description
IsSystem	boolean	True if in system model

6.2.2. System.GroupProperties

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

Column Name	Type	Description
ModelName	string	Model name
GroupFullName	string	Full group name
Name	string	Property name
Value	string	Property value
GroupName	string	Short group name
GroupUpperName	string	Full upper-case group name
ID	string	Group unique ID

6.2.3. System.Elements

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

Column Name	Type	Description
ModelName	string	Model name
GroupName	string	Short group name
GroupFullName	string	Full group name
Name	string	Element name (not qualified)
Position	integer	Position in group (1-based)
NameInSource	string	Name of element in source
DataType	string	MetaMatrix 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

Column Name	Type	Description
IsSigned	boolean	Element is signed numeric value
IsCurrency	boolean	Element represents monetary value
IsAutoIncremented	boolean	Element is auto-incremented in the source
NullType	string	Nullability: "Nullable", "No Nulls", "Unknown"
MinRange	string	Minimum numeric value
MaxRange	string	Maximum numeric value
SearchType	string	Searchability: "Searchable", "All Except Like", "Like Only", "Unsearchable"
Format	string	Format of string value
DefaultValue	string	Default value
JavaClass	string	Java class that will be returned
Precision	integer	Number of digits in numeric value
CharOctetLength	integer	Measure of return value size
Radix	integer	Radix for numeric values
GroupUpperName	string	Upper-case full group name
UpperName	string	Upper-case element name
UID	string	Element unique ID
Description	string	Description

6.2.4. System.ElementProperties

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

Column Name	Type	Description
ModelName	string	Model name
GroupFullName	string	Full group name
ElementName	string	Element name
Name	string	Property name
Value	string	Property value
GroupName	string	Short group name

Column Name	Type	Description
ElementUpperName	string	Upper-case element name
GroupUpperName	string	Upper-case group name
UID	string	Element unique ID

6.2.5. System.Keys

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

Column Name	Type	Description
ModelName	string	Model name
GroupFullName	string	Full group 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
GroupName	string	Short group name
GroupUpperName	string	Upper-case full group name
RefKeyUID	string	Referenced key UID (if foreign key)
UID	string	Key unique ID

6.2.6. System.KeyProperties

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

Column Name	Type	Description
Column Name	Type	Description
ModelName	string	Model name
GroupFullName	string	Full group name
KeyName	string	Key name
Name	string	Extension property name
Value	string	Extension property value
GroupName	string	Short group name
GroupUpperName	string	Upper-case full group name
UID	string	Key unique ID

6.2.7. System.KeyElements

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

Column Name	Type	Description
ModelName	string	Model name
GroupFullName	string	Full group name
Name	string	Element name
KeyName	string	Key name
KeyType	string	Key type: "Primary", "Foreign", "Unique", etc
GroupName	string	Short group name
GroupUpperName	string	Upper case full group name
RefKeyUID	string	Referenced key UID
UID	string	Key UID
Position	integer	Position in key

6.3. Procedure Metadata

6.3.1. System.Procedures

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

Column Name	Type	Description
ModelName	string	Model name
Name	string	Procedure name
NameInSource	string	Procedure name in source system
ReturnsResults	boolean	Returns a result set
ModelUID	string	Model UID
UID	string	Procedure UID
Description	string	Description
FullName	string	Full procedure name

6.3.2. System.ProcedureProperties

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

Column Name	Type	Description
ModelName	string	Model name

Column Name	Type	Description
ProcedureName	string	Procedure name
Name	string	Property name
Value	string	Property value
UID	string	Procedure UID

6.3.3. System.ProcedureParams

This supplies information on procedure parameters.

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

6.4. Datatype Metadata

6.4.1. System.DataTypes

This table supplies information on *datatypes*.

Column Name	Type	Description
Name	string	MetaMatrix design-time type name
IsStandard	boolean	Always false
IsPhysical	boolean	Always false

Column Name	Type	Description
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
RuntimeType	string	MetaMatrix runtime data type name
BaseType	string	Base type
Description	string	Description of type

6.4.2. System.DataTypeProperties

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

Column Name	Type	Description
DataType	string	Data type name
Name	string	Property name
Value	string	Property value
UID	string	Data type UID

Federated Planning

MetaMatrix 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.

7.1. 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 MetaMatrix 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, MetaMatrix also takes advantage of costing information. The logical plan optimization steps can be seen by using the *OPTION DEBUG* clause.
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*.

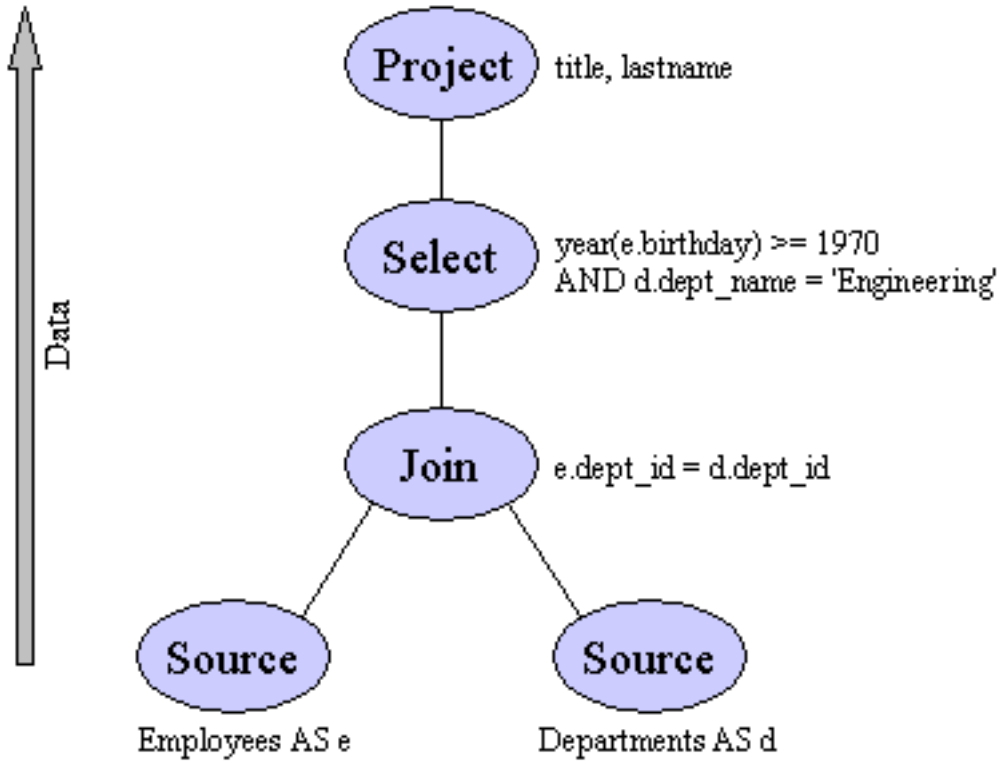
The logical query plan is a tree of operations used to transform data in source tables to the expected result set. In the tree, data flows from the bottom (tables) to the top (output). The primary logical operations are *select* (select or filter rows based on a criteria), *project* (project or compute 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 7.1. 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.

7.2. Federated Optimizations

7.2.1. 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 MetaMatrix though the Connector API. Any work not performed at the source is then processed in MetaMatrix's relational engine.

Based upon capabilities, MetaMatrix 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.

7.2.2. Dependent Joins

A special optimization called a dependent join is used to reduce the rows returned from one of the two relations involved in a multi-source join. In a dependent join, queries are issued to each source sequentially rather than in parallel, with the results obtained from the first source used to restrict the records returned from the second. Dependent joins can perform some joins much faster by drastically reducing the amount of data retrieved from the second source and the number of join comparisons that must be performed.

The conditions when a dependent join is used are determined by the query planner based on access patterns, hints, and costing information.

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

Tip

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

7.2.3. 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

7.2.4. Projection Minimization

MetaMatrix 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.

7.2.5. Partial Aggregate Pushdown

Partial aggregate pushdown allows for grouping operations above multi-source joins to be decomposed so that some of the grouping and aggregate functions may be pushed down to the sources.

7.2.6. Optional Join

The optional join hint indicates to the MetaMatrix optimizer that a join clause should be omitted if none of its columns are used in either user criteria or output columns in the result. This hint is typically only used in view layers containing multi-source joins.

The optional join hint is applied as a comment on a join clause.

Example 7.2. Example Optional Join Hint

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

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

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

Tip

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

7.2.7. Standard Relational Techniques

MetaMatrix 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.

7.3. Federated Failure Modes

7.3.1. Partial Results

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

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

7.3.1.1. Setting Partial Results Mode

Partial results mode is off by default but can be turned on by default for all queries in a Connection with either setPartialResultsMode("true") on a DataSource or partialResultsMode=true on a JDBC URL. In either case, partial results mode may be overridden on a per-query basis by setting the execution property on the Statement. To set this property, cast to the MetaMatrix Statement JDBC API extension interface `com.metamatrix.jdbc.api.Statement`

Example 7.3. Example - Setting Partial Results Mode

```
Statement statement = ...obtain statement from Connection...

com.metamatrix.jdbc.api.Statement mmStatement =
  (com.metamatrix.jdbc.api.Statement) statement;

mmStatement.setExecutionProperty(
  ExecutionProperties.PROP_PARTIAL_RESULTS_MODE, "true");
```

This property can be set before each execution (via an execute method) on a Statement (or PreparedStatement or CallableStatement).

7.3.1.2. Source Unavailability

A source is considered to be 'unavailable' if the connector binding associated with the source issues an exception in response to a query. The exception will be propagated to the query processor, where it will become a warning in the result set.

Warning

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

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

Example 7.4. Example - Printing List of Failed Sources

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

7.4. Query Plans

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

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

7.4.1. Getting a Query Plan

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

- OPTION SHOWPLAN - Returns the plan in addition to any results
- OPTION PLANONLY - Returns the plan, does not execute the command though

With the above options, the query plan is available from the Statement object by casting to the `com.metamatrix.jdbc.api.Statement` interface.

Example 7.5. Retrieving a Query Plan

```
ResultSet rs = statement.executeQuery("select ...");
com.metamatrix.jdbc.api.Statement mmstatement =
 (com.metamatrix.jdbc.api.Statement)statement;
PlanNode queryPlan = mmstatement.getPlanDescription();
System.out.println(XMLOutputVisitor.convertToXML(queryPlan);
```

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

7.4.2. Analyzing a Query Plan

Once a query plan has been obtained you will most commonly be looking for:

- Source pushdown -- what parts of the query that got pushed to each source
- Join ordering
- Join algorithm used - merge or nested loop.
- Presence of federated optimizations, such as dependent joins.
- Join criteria type mismatches.

All of these issues presented above will be present subsections of the plan that are specific to relational queries. If you are executing a procedure or generating an XML document, the overall query plan will contain additional information related the surrounding procedural execution.

A query plan consists of a set of nodes organized in a tree structure. As with the above example, you will typically be interested in analyzing the textual form of the plan.

In a procedural context the ordering of child nodes implies the order of execution. In most other situation, child nodes may be executed in any order even in parallel. Only in specific optimizations, such as dependent join, will the children of a join execute serially.

7.4.3. Relational Plans

Relational plans represent the actually processing plan that is composed of nodes that are the basic building blocks of logical relational operations. Physical relational plans differ from logical relational plans in that they will contain additional operations and execution specifics that were chosen by the optimizer.

The nodes for a relational query plan are:

- Access - Access a source. A source query is sent to the connector binding 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.

- Dependent Feeder - This node accepts its input stream and forwards to its parent unchanged but also feeds all dependent sources that need the stream of data. Thus, this node actually performs no work within the tree, just diverts a copy of the tuple stream to listening nodes.

- Dependent Wait - This node waits until a criteria requiring dependent values below this node has the necessary data to continue. At that point, it continues processing on it's subplan and merely forwards data from the child to the parent.

7.4.3.1. Node Statistics

Every node has a set of statistics that are output. These can be used to determine the amount of data flowing through the node.

Statistic	Description	Units
Node Output Rows	Number of records output from the node	count
Node Process Time	Time processing in this node only	millisec
Node Cumulative Process Time	Elapsed time from beginning of processing to end	millisec
Node Cumulative Next Batch Process Time	Time processing in this node + child nodes	millisec
Node Next Batch Calls	Number of times a node was called for processing	count
Node Blocks	Number of times a blocked exception was thrown by this node or a child	count

In addition to node statistics, some nodes display cost estimates computed at the node.

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

Architecture

8.1. Data Management

8.1.1. Buffer Management

The buffer manager manages memory for ALL result sets used in the query engine . That includes result sets read from a connector binding , 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.

8.1.2. 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.

8.1.3. 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 NOT removed from the file when they are swapped back into memory because that would require removing data out of the middle of the file and updating all the indexes which would be very expensive. Thus the disk storage manager never removes a particular batch. 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 - customers may want to increase the number of open files allowed (a configuration parameter defaulted to 10).

8.1.4. 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. This is a final cleanup mechanism that removes all state associated with a 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. In general, these

mechanisms mean that the engine should always shut down with 0 open files. 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.

8.2. Multi-Source Cursoring

MetaMatrix 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.

MetaMatrix processes results in batches. A batch is simply a set of records. The size of the batches processed in the query engine is configurable on system startup.

When a batch has been filled, it is available to be returned to the requesting client. The first batch is always automatically returned. Subsequent batches are returned based on client demand for the data. Client applications have no knowledge of batches or batch sizes. They merely request the ‘next’ record, and if the record is not currently available on the client-side, it is fetched from the server-side. Pre-fetching of batches is performed immediately on receipt of the prior batch. Data can return to the JDBC API asynchronously to the client's use of the API, possibly making data always available (depends on the timing of the client application and query).

8.3. Cancelling Queries

If the client issues a ‘cancel’ command, then no results from the batch currently being processed in the server will be returned to the client. This is true for both the first batch and subsequent batches.

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. JDBC connectors will asynchronously call cancel on the underlying JDBC driver, which may or may not actually support this method.

8.4. Timeouts

Timeouts in MetaMatrix are managed on the client-side, in the JDBC API. Timeouts are only relevant for the first record returned. If the first record has not been received by the client within the specified timeout period, a ‘cancel’ command is issued to the server for the request and no results are returned to the client. The cancel command is issued by the JDBC API without the client’s intervention.

Note that any results in the first batch that are pending in the server will not be returned to the client when the timeout/cancel occurs.

Appendix A. BNF Grammar

A.1. Terminals

<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">| <BOOLEAN: "boolean">| <BYTE: "byte">| <SHORT: "short">| <CHAR: "char">| <INTEGER: "integer">| <LONG: "long">| <BIGINTEGER: "biginteger">| <FLOAT: "float">| <DOUBLE: "double">| <BIGDECIMAL: "bigdecimal">| <DATE: "date">| <TIME: "time">| <TIMESTAMP: "timestamp">| <OBJECT: "object">| <BLOB: "blob">| <CLOB: "clob">| <XML: "xml">}
<DEFAULT> TOKEN : {<CAST: "cast">| <CONVERT: "convert">| <TIMESTAMPADD: "timestampadd">| <TIMESTAMPDIFF: "timestampdiff">| <COUNT: "count">| <SUM: "sum">| <AVG: "avg">| <MIN: "min">| <MAX: "max">}
<DEFAULT> TOKEN : {<ALL: "all">| <AND: "and">| <ANY: "any">| <AS: "as">| <ASC: "asc">| <BEGIN: "begin">| <BETWEEN: "between">| <BREAK: "break">| <BY: "by">| <CASE: "case">| <CONTINUE: "continue">| <CREATE: "create">| <CRITERIA: "criteria">| <CROSS: "cross">| <DEBUG: "debug">| <DECLARE: "declare">| <DELETE: "delete">| <DESC: "desc">| <DISTINCT: "distinct">| <DROP: "drop">| <ELSE: "else">| <END: "end">| <ERROR: "error">| <ESCAPE: "escape">| <EXCEPT: "except">| <EXEC: "exec">| <EXECUTE: "execute">| <EXISTS: "exists">| <FALSE: "false">| <FN: "fn">| <FOR: "for">| <FROM: "from">| <FULL: "full">| <GROUP: "group">| <HAS: "has">| <HAVING: "having">| <IF: "if">| <IN: "in">| <INNER: "inner">| <INSERT: "insert">| <INTERSECT: "intersect">| <INTO: "into">| <IS: "is">| <JOIN: "join">| <LEFT: "left">| <LIKE: "like">| <LIMIT: "limit">| <LOCAL: "local">| <LOOP: "loop">| <MAKEDEP: "makedep">| <MAKENOTDEP: "makenotdep">| <NOCACHE: "nocache">| <NOT: "not">| <NULL: "null">| <ON: "on">| <OJ: "oj">| <OPTION: "option">| <OR: "or">| <ORDER: "order">| <OUTER: "outer">| <PLANONLY: "planonly">| <PROCEDURE: "procedure">| <RIGHT: "right">| <SELECT: "select">| <SET: "set">| <SHOWPLAN: "showplan">| <SOME: "some">| <TABLE: "table">| <TEMPORARY: "temporary">| <THEN: "then">| <TRANSLATE: "translate">| <TRUE: "true">| <UNION: "union">| <UNKNOWN: "unknown">| <UPDATE: "update">| <USING: "using">| <VALUES: "values">| <VIRTUAL: "virtual">| <WHEN: "when">| <WHERE: "where">| <WITH: "with">| <WHILE: "while">}
<DEFAULT> TOKEN : {<SQL_TSI_FRAC_SECOND: "SQL_TSI_FRAC_SECOND">| <SQL_TSI_SECOND: "SQL_TSI_SECOND">| <SQL_TSI_MINUTE: "SQL_TSI_MINUTE">| <SQL_TSI_HOUR: "SQL_TSI_HOUR">| <SQL_TSI_DAY: "SQL_TSI_DAY">| <SQL_TSI_WEEK: "SQL_TSI_WEEK">| <SQL_TSI_MONTH: "SQL_TSI_MONTH">| <SQL_TSI_QUARTER: "SQL_TSI_QUARTER">| <SQL_TSI_YEAR: "SQL_TSI_YEAR">}
<DEFAULT> TOKEN : {<ALL_IN_GROUP: (<GROUP_PART> | <MMUUID_PART>) <PERIOD> <STAR>>| <VARIABLE: <ID> | <MMUUID>>| <#ID: <GROUP_PART> ((<PERIOD>

| <SLASH>) (<QUOTED_ID> | <MMUUID_PART>))?>| <#ELEMENT: <GROUP_PART> (<PERIOD> | <SLASH>) <QUOTED_ID>>| <#GROUP_PART: ("#")? (<QUOTED_ID> (<PERIOD> | <SLASH>))? <QUOTED_ID>>| <#QUOTED_ID: <DOTTED_ID> | "\"" <DOTTED_ID> "\"">| <#DOTTED_ID: <ID_PART> ((<PERIOD> | <SLASH>) <ID_PART>)*>| <#ID_PART: ("@")? <LETTER> (<ID_CHAR>)*>| <#ID_CHAR: <LETTER> | "_" | <DIGIT>>| <#MMUUID: <MMUUID_PART> (<PERIOD> <MMUUID_PART>)?>| <#MMUUID_PART: "mmuuid:" (<MMUUID_CHAR>)*>| <#MMUUID_CHAR: ["a"-"f"] | ["0"-"9"] | "-">| <DATETYPE: "{" "d">| <TIMETYPE: "{" "t">| <TIMESTAMPTYPE: "{" "ts">| <BOOLEANTYPE: "{" "b">| <INTEGERVAL: (<MINUS>)? (<DIGIT>)+>| <FLOATVAL: (<MINUS>)? (<DIGIT>)* <PERIOD> (<DIGIT>)+ (["e","E"] (["+","-"])? (<DIGIT>)+)?>| <STRINGVAL: ("N")? (<STRINGA> | <STRINGB>)>| <#STRINGA: "\"" (~["\""])* ("\"\"" (~["\""])*)* "\"">| <#STRINGB: "\"" (~["\""])* ("\"\"" (~["\""])*)* "\"">| <#LETTER: ["a"-"z","A"-"Z"] | ["\u0153"-"\ufffd"]>| <#DIGIT: ["0"-"9"]>| <#COLON: ":">}

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

A.2. Non-Terminals

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

	((<EQ> | <NE> | <NE2> | <LE> | <GE> | <LT> | <GT> | <IN> | <LIKE> | (<IS> <NULL>) | <BETWEEN>))? <CRITERIA> (<ON> <LPAREN> <VARIABLE> (<COMMA> <VARIABLE>)* <RPAREN>)?
hasCriteria ::=	<HAS> *criteriaSelector*
declareStatement ::=	<DECLARE> *dataType* <VARIABLE> (<EQ> *assignStatementOperand*)?
assignStatement ::=	<VARIABLE> <EQ> *assignStatementOperand*
assignStatementOperand ::=	((*insert*) | *update* | *delete* | *storedProcedure* | (*expression*) | *setQuery*)
sqlStatement ::=	(*setQuery* | (*dynamicCommand*) | *storedProcedure* | *insert* | *update* | *delete* | *createTempTable* | *dropTable*)
translateCriteria ::=	<TRANSLATE> *criteriaSelector* (<WITH> <LPAREN> <VARIABLE> <EQ> *expression* (<COMMA> <VARIABLE> <EQ> *expression*)* <RPAREN>)?
createUpdateProcedure ::=	<CREATE> (<VIRTUAL>)? (<UPDATE>)? <PROCEDURE> *block*
dynamicCommand ::=	<EXECUTE> <STRING> *expression* (<AS> *createElementsWithTypes* (<INTO> <VARIABLE>)?)? (<USING> <VARIABLE> <EQ> (<COMMA> <VARIABLE> <EQ>)*)? (<UPDATE> ((<INTEGERVAL>) | (<STAR>)))?
createElementsWithTypes ::=	<VARIABLE> *dataType* (<COMMA> <VARIABLE> *dataType*)*
storedProcedure ::=	((<EXEC> | <EXECUTE>) <VARIABLE> <LPAREN> (*executeNamedParams* | *executeUnnamedParams*) <RPAREN>) (*option*)?
executeUnnamedParams ::=	(*expression* (<COMMA> *expression*)*)?
executeNamedParams ::=	(*paramName* <EQ> *expression* (<COMMA> *paramName* <EQ> *expression*)*)
paramName ::=	<VARIABLE>
insert ::=	<INSERT> <INTO> <VARIABLE> (<LPAREN> <VARIABLE> (<COMMA> <VARIABLE>)* <RPAREN>)? ((<VALUES> *rowValues*) | (*setQuery*)) (*option*)?
rowValues ::=	<LPAREN> *expression* (<COMMA> *expression*)* <RPAREN>
update ::=	<UPDATE> <VARIABLE> <SET> <VARIABLE> <EQ> *expression* (<COMMA> <VARIABLE> <EQ> *expression*)* (*where*)? (*option*)?

delete ::=	<DELETE> <FROM> <VARIABLE> (*where*)? (*option*)?
setQuery ::=	*basicQuery* ((<UNION> (<ALL>)? *basicQuery*)+)? (*orderby*)? (*limit*)? (*option*)?
basicQuery ::=	(*query* | (<LPAREN> *setQuery* <RPAREN>))
query ::=	*select* (*into*)? (*from* (*where*)? (*groupBy*)? (*having*)?)?
into ::=	<INTO> (<VARIABLE>)
select ::=	<SELECT> (<ALL> | (<DISTINCT>))? (<STAR> | (*selectSymbol* (<COMMA> *selectSymbol*)*))
selectSymbol ::=	((<ALL_IN_GROUP>) | (*expression*) ((<AS>)? (<VARIABLE> | <STRINGVAL>))?)
aggregateSymbol ::=	((<COUNT> <LPAREN> <STAR> <RPAREN>) | ((<COUNT> | <SUM> | <AVG> | <MIN> | <MAX>) <LPAREN> (<DISTINCT>)? *expression* <RPAREN>))
from ::=	<FROM> (*fromClause* (<COMMA> *fromClause*)*)
fromClause ::=	((<LBRACE> <OJ> *fromClauseUnescaped* <RBRACE>) | *fromClauseUnescaped* | (<EXEC> <LPAREN> *fromClauseUnescaped* <RPAREN>))
fromClauseUnescaped ::=	*basicFromClause* ((((<CROSS> <JOIN> | <UNION> <JOIN>) *fromClauseUnescaped*) | ((<RIGHT> (<OUTER>)? | <LEFT> (<OUTER>)? | <FULL> (<OUTER>)? | <INNER>)? <JOIN> *fromClauseUnescaped* <ON> *criteria*)))*
basicFromClause ::=	(*unaryFromClause* | *subqueryFromClause* | (<LPAREN> *fromClause* <RPAREN>)) ((<MAKEDEP>) | (<MAKENOTDEP>))?
subqueryFromClause ::=	<LPAREN> (*setQuery* | *storedProcedure*) <RPAREN> (<AS>)? <VARIABLE>
unaryFromClause ::=	(<VARIABLE> ((<AS>)? <VARIABLE>)?)
where ::=	<WHERE> *criteria*
criteria ::=	*compoundCritOr*
compoundCritOr ::=	*compoundCritAnd* (<OR> *compoundCritAnd*)*
compoundCritAnd ::=	*notCrit* (<AND> *notCrit*)*
notCrit ::=	(<NOT>)? *primary*
primary ::=	(*predicate* | (<LPAREN> *criteria* <RPAREN>))
predicate ::=	(*subqueryCompareCriteria* | *compareCrit* | *matchCrit* | *betweenCrit* | *setCrit* | *existsCriteria* | *hasCriteria* | *translateCriteria* | *isNullCrit*)

compareCrit ::=	*expression* (<EQ> | <NE> | <NE2> | <LT> | <LE> | <GT> | <GE>) *expression*
subquery ::=	<LPAREN> (*setQuery* | *storedProcedure*) <RPAREN>
subqueryCompareCriteria ::=	*expression* (<EQ> | <NE> | <NE2> | <LT> | <LE> | <GT> | <GE>) (<ANY> | <SOME> | <ALL>) *subquery*
matchCrit ::=	(*expression* (<NOT>)? <LIKE> *expression* ((<ESCAPE> <STRINGVAL>) | (<LBRACE> <ESCAPE> <STRINGVAL> <RBRACE>))?)
betweenCrit ::=	*expression* (<NOT>)? <BETWEEN> *expression* <AND> *expression*
isNullCrit ::=	*expression* <IS> (<NOT>)? <NULL>
setCrit ::=	*expression* (<NOT>)? <IN> ((*subquery*) | (<LPAREN> *expression* (<COMMA> *expression*)* <RPAREN>))
existsCriteria ::=	<EXISTS> *subquery*
groupBy ::=	<GROUP> <BY> (*groupByItem* (<COMMA> *groupByItem*)*)
groupByItem ::=	*expression*
having ::=	<HAVING> *criteria*
orderby ::=	<ORDER> <BY> (<VARIABLE> | <STRINGVAL> | <INTEGERVAL>) (<ASC> | <DESC>)? (<COMMA> (<VARIABLE> | <STRINGVAL> | <INTEGERVAL>) (<ASC> | <DESC>)?)*
limit ::=	<LIMIT> (<INTEGERVAL> | <QMARK>) (<COMMA> (<INTEGERVAL> | <QMARK>))?
option ::=	<OPTION> (<SHOWPLAN> | <PLANONLY> | <DEBUG> | <MAKEDEP> <VARIABLE> (<COMMA> <VARIABLE>)* | <MAKENOTDEP> <VARIABLE> (<COMMA> <VARIABLE>)* | <NOCACHE> (<VARIABLE> (<COMMA> <VARIABLE>)*)?)*
expression ::=	*concatExpression*
concatExpression ::=	(*plusExpression* (<CONCAT_OP> *plusExpression*)*)
plusExpression ::=	(*timesExpression* (*plusOperator timesExpression*)*)
plusOperator ::=	(<PLUS> | <MINUS>)
timesExpression ::=	(*basicExpression* (*timesOperator basicExpression*)*)
timesOperator ::=	(<STAR> | <SLASH>)
basicExpression ::=	(<QMARK> | *literal* | (<LBRACE> <FN> *function* <RBRACE>) | (*aggregateSymbol*) | (*function*) | (<VARIABLE>

	) | (<LPAREN> *expression* <RPAREN>) | *subquery* | *caseExpression* | *searchedCaseExpression*)
caseExpression ::=	<CASE> *expression* (<WHEN> *expression* <THEN> *expression*)+ (<ELSE> *expression*)? <END>
searchedCaseExpression ::=	<CASE> (<WHEN> *criteria* <THEN> *expression*)+ (<ELSE> *expression*)? <END>
function ::=	((<CONVERT> <LPAREN> *expression* <COMMA> *dataType* <RPAREN>) | (<CAST> <LPAREN> *expression* <AS> *dataType* <RPAREN>) | ((<TIMESTAMPADD> | <TIMESTAMPDIFF>) <LPAREN> *intervalType* <COMMA> *expression* <COMMA> *expression* <RPAREN>) | ((<LEFT> | <RIGHT> | <CHAR>) <LPAREN> (*expression* (<COMMA> *expression*)*)? <RPAREN>) | ((<INSERT>) <LPAREN> (*expression* (<COMMA> *expression*)*)? <RPAREN>) | ((<TRANSLATE>) <LPAREN> (*expression* (<COMMA> *expression*)*)? <RPAREN>) | (<VARIABLE> <LPAREN> (*expression* (<COMMA> *expression*)*)? <RPAREN>))
dataType ::=	(<STRING> | <BOOLEAN> | <BYTE> | <SHORT> | <CHAR> | <INTEGER> | <LONG> | <BIGINTEGER> | <FLOAT> | <DOUBLE> | <BIGDECIMAL> | <DATE> | <TIME> | <TIMESTAMP> | <OBJECT> | <BLOB> | <CLOB> | <XML>)
intervalType ::=	(<SQL_TSI_FRAC_SECOND> | <SQL_TSI_SECOND> | <SQL_TSI_MINUTE> | <SQL_TSI_HOUR> | <SQL_TSI_DAY> | <SQL_TSI_WEEK> | <SQL_TSI_MONTH> | <SQL_TSI_QUARTER> | <SQL_TSI_YEAR>)
literal ::=	(<STRINGVAL> | <INTEGERVAL> | <FLOATVAL> | <FALSE> | <TRUE> | <NULL> | (<BOOLEANTYPE> <STRINGVAL> <RBRACE>) | (<TIMESTAMPTYPE> <STRINGVAL> <RBRACE>) | (<DATETYPE> <STRINGVAL> <RBRACE>) | (<TIMETYPE> <STRINGVAL> <RBRACE>))