Matisse® SQL Programmer's Guide

January 2017

Matissse SQL Programmer’s Guide

PDF generated 7 January 2017

Tables

Contents

Introduction

This manual describes the syntax and usage of the Matisse SQL language. Matisse SQL allows you to write reusable server components to build application business logic, select a set of objects that meet certain criteria regarding the attribute values or the relationships, and update objects.

Conventions

This document uses the following conventions:

Text

The running text is written in characters like these.

`Code`

All computer variables, code, commands, and interactions are shown in this font.

`variable`

In a program example, or in an interaction, a variable, which means anything that is dependent on the user environment, is written in italics.

`KEYWORD`

In syntax descriptions, an SQL keyword always appears in uppercase Courier.

`{ANY|ALL}`

In syntax descriptions, curly braces are used to enclose two or more choices among different keywords or expressions. The choices themselves are separated by a vertical bar |.

`[id|keyword]`

In syntax descriptions, brackets are used to enclose one or more optional keywords or expressions. If there are two or more choices, they are separated by a vertical bar |, and you can specify only one.

References

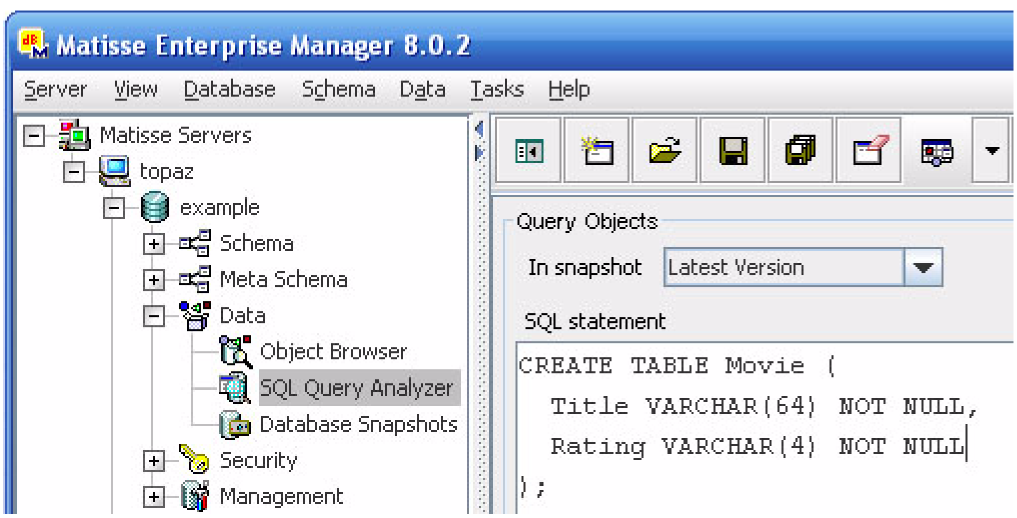
References to another part of the Matisse documentation are made as shown here.

1 SQL Query Analyzer and mt_sql Utility

The SQL Query Analyzer in the Matisse Enterprise Manager is the graphical environment which allows you to execute SQL statements and define SQL methods (stored procedures). The `mt_sql` utility is Matisse's command-line interface allowing you to interactively execute SQL statements and display the result.

1.1 SQL Query Analyzer

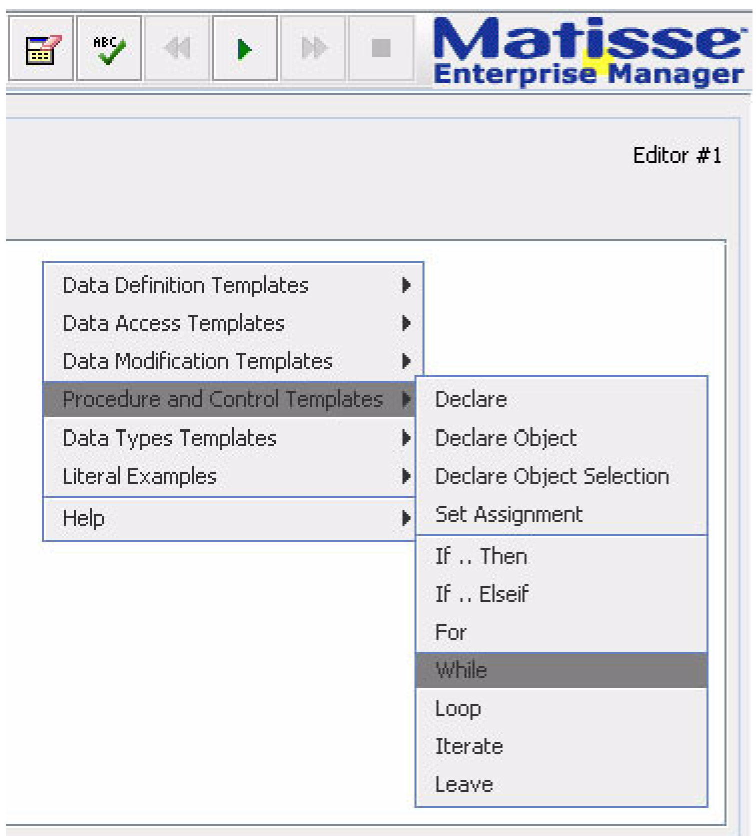
Start the Enterprise Manager (double click the Enterprise Manager icon, start-->Programs-->Matisse-->Enterprise Manager, or type mt_emgr from a command line), select a database, start the database if it has not started yet, open the 'Data' node, then select 'SQL Query Analyzer'.



Enter SQL statement(s), and click the execute button to execute the statement(s).

You can enter a single SQL statement or multiple SQL statements. For multiple statements, each statement needs to be terminated with a semi-colon “;”.

Online Help is available for SQL statement’s syntax, types, program controls, and templates. Right-click in the SQL query editor window:



For more information about the Matisse Enterprise Manager, refer to the "Discovering Matisse Enterprise Manager" document which is accessible from the Matisse Readme file (readme.html) or the Matisse Server Administration Guide.

1.2 Simple Example with mt_sql

The following is a simple example of using the mt_sql utility for creating a class, inserting and accessing objects:

```
% mt_sql -d my_db@my_host
sql> CREATE CLASS movie (
      title STRING,
      rating STRING
);
sql> COMMIT;
sql> INSERT INTO movie (title, rating)
     VALUES ('Rocky', 'R');
sql> COMMIT;
sql> SELECT * FROM movie;
OID                title               rating
------------------ ------------------- -------------------
0x1047             Rocky               R
1 objects selected
sql> quit;
```

More details are explained in the following sections.

1.3 Basic Usage

An SQL statement can be a single line or can be divided into multiple lines. It must be terminated by a semicolon (;) in either case. For example,

```
sql> SELECT lastName, firstName
  2>  FROM artist
  3>  WHERE lastName LIKE 'S%';
```

You can exit `mt_sql` with the command `quit`:

```
sql> quit;
```

If you execute an SQL statement and no transaction or read-only access is started explicitly, `mt_sql` starts a read-only access to the latest version of the database. When the SQL statement execution is done, `mt_sql` terminates the read-only access immediately.

If you start a transaction or a read-only access explicitly using:

```
SET TRANSACTION READ {WRITE | ONLY}
```

then `mt_sql` keeps the transaction or read-only access open until you commit or abort the transaction, or end the read-only access. Note that you cannot update both the schema and other database objects in the same transaction. The following statements need to be executed in different transactions, since the first statement is creating schema objects, i.e., classes, attributes, while the following INSERT statement creates a regular object:

```
% mt_sql -d my_db@my_host
sql> SET TRANSACTION READ WRITE;
Transaction read write started 0
sql> CREATE CLASS movie (
  2>   title STRING,
  3>   rating STRING
  4> );
Class "movie" created
sql> COMMIT;
Transaction commited
sql> SET TRANSACTION READ WRITE;
Transaction read write started 0
sql> INSERT INTO movie (title, rating)
  2>   VALUES ('Rocky', 'PG');
1 object inserted
sql> COMMIT;
Transaction commited
```

1.4 Command Line Options

The `mt_sql` utility can take several options. The `-h` option gives you a simple explanation for all the options as listed in *Table 1.1*.

```
Usage: mt_sql [-d [user:]dbname[@host[:port]]] [-qopshV]
```

Table 1.1 Command Line Options

Option	Explanation
`-d, --database=...`	Specify the database and host in the format of dbname@host
`-q, --quiet`	When you specify this option, no output is printed on your terminal. The sql> prompt is not shown either.
`-V, --version`	Print the version of the utility and exit.
`-p, --passwd=...`	Specify the password to connect to the database.
`-s, --size=...`	Display size for string types (default 20)
`-h, --help`	Display this help and exit.

When you write a statement with BEGIN and END, such as a CREATE METHOD statement, BEGIN and END must be the only word in a line. For example:

```
sql> CREATE METHOD foo ()
> RETURNS INTEGER
> FOR class_foo
> BEGIN
>   ...
> END;
```

1.5 Online Help

The utility has an online help that provides you with a simple description for each SQL command, keyword, or built-in function.

To see a summary of available help commands, type “help”.

```
sql> help;
```

To see a description of each command, type “help <command>”. For example,

```
sql> help set transaction;
```

then you will see:

```
SyntaxSET TRANSACTION READ
      {ONLY [<version>]
      |WRITE [<priority>]}
Purpose:Start a version access (read-only transaction) or a
        transaction.
    ...
```

1.6 Discovering the Schema

You can discover a database schema using SQL statements.

1. Getting the names of all the classes:

```
sql> SELECT MtName FROM MtClass;
MtName
--------------------
movie
...
```

2. Getting the names of the attributes defined in a class:

```
sql> SELECT MtAttributes.MtName FROM MtClass
  2>   WHERE MtName = 'movie';
MtName
--------------------
title
rating
```

A quicker way to discover all the attribute names is to use a `SELECT` statement that selects no object:

```
sql> SELECT * FROM movie WHERE 1 = 2;
title                 rating
--------------------  --------------------
0 objects selected
```

3. Getting the names of the relationships defined in a class:

```
sql> SELECT MtRelationships.MtName FROM MtClass
  2>   WHERE MtName = 'movie';
MtName
--------------------
directedBy
starring
```

2 Constants and Identifiers

This section describes the different elements of a Matisse SQL command. The elements that make up a request are separated by at least one separator. A separator can be a blank space, a tab, or a carriage return.

After reading this section you should be familiar with:

- Constants
- Identifiers
- Keywords

2.1 What Is a Constant?

A *constant* is a value of one of the following types:

- Integer number
- Numeric number
- Real number
- Boolean
- Character string
- Null value
- Timestamp
- Date
- Time interval
- Bytes
- List of all the above types, except null and bytes.

Note that an undetermined value is expressed by the keyword `NULL`.

Integer Constants

An *integer constant* is a string of 19 numerals at the most. It does not contain spaces, and may be preceded by a plus + or a minus –. Maximum and minimum values are 9223372036854775807 and -9223372036854775808, respectively. Here is examples:

```
12
-123456879
```

Numeric Constants

A numeric constants is a combination of integer number constants and a decimal point ".", and may be preceeded by a plus + or a minus - sign. For example:

```
12.34
  -.1
```

This type has a precision and scale. The scale is the number of digits in the fractional part of the number, and cannot be negative or greater than the precision.

Real Constants

A *real number constant*, an approximate number, is a combination of integer number constants and keywords "." and "E" (or "e"). It can take the following forms, where "x" represents an integer:

```
x
.x
x.
x.x
x.E[+-]x
.xE[+-]x
x.xE[+-]x
```

The following examples show real number constants that are valid:

```
12.
-.2
+143.5e-4
```

Boolean Constants

You can declare boolean attributes in the Matisse database schema with the type `BOOLEAN`. In Matisse SQL, boolean constants can take one of the two values:

```
TRUE
FALSE
```

For instance, to check if a boolean attribute `MARRIED` is set to `TRUE` you can write the following predicate in a where-clause:

```
MARRIED = TRUE
```

Character String Constants

A *character string constant* is a string of characters that does not include carriage returns or non-printable characters, enclosed by single quotes. A character string constant can be empty.

```
'this is a text string'
''
```

Matisse recognizes several escape sequences within strings that indicate special characters. Each sequence begins with a backslash character ('\') to signify a temporary escape from the usual rules for character interpretation. `\b` is a backspace, `\f` is a form feed, `\n` is a newline, `\r` is a carriage return, `\t` is a tab. Thus, to include a backslash in a string constant, type two backslashes.

```
'another text.\r\n'
```

You can specify unicode character string constants for `UTF16` using a `ASCII` character or a escape sequence `\uXXXX` that represents a `UTF16-LE` (Little Endian) character by specifying the `N` keyword as follows:

```
N 'a text with unicode sequences \u00E9\u00F1'
```

You can specify unicode character string constants for `UTF8` by specifying the `UTF8` keyword as follows:

```
UTF8 'another unicode text'
```

You can specify a single quote in a character string constant, by specifying two contiguous quotes. In the definition of a character string constant, two contiguous quotes have a length of one character. The following example shows how to enter a character string containing a single quote:

```
'Computer''s disks'
```

Note that the above character string has a length of 16.

Character strings are case sensitive.

Date and Timestamp Constants

A *date constant* is expressed with the following syntax:

```
DATE 'yyyy-mm-dd'
```

Where `yyyy-mm-dd` represents respectively the year with 4 digits, and the month and day of the month with 2 digits.

For instance, if you want to check for the value of an attribute `birthdate` to retrieve objects with a birth date later that October 10, 2007, you could write the following predicate:

```
birthdate > DATE '2007-10-10'
```

To get the current date, use the following:

```
CURRENT_DATE()
```

A *timestamp constant* is expressed with the following syntax:

```
TIMESTAMP 'yyyy-mm-dd hh:mm:ss[.uuuuuu]'
    [ AT {LOCAL | GMT | UTC}]
```

To the date specification is added *`hh:mm:ss`* that represents respectively the hour, minutes and seconds, each using 2 digits. An optional fraction of seconds can be specified up to 6 digits.

For instance, if we suppose that we run an application where each operation updates a `lastEntry` attribute, you could check for the objects where the last entry was entered after October 1, 2007 at 11:30 AM with the following predicate:

```
lastEntry >
```

```
    TIMESTAMP '2007-10-01 11:30:00'
```

The two following expressions are also valid and lead to the same result:

```
lastEntry >
    TIMESTAMP '2007-10-01 11:30:00.00'
lastEntry >
    TIMESTAMP '2007-10-01 11:30:00.000000'
```

By default the `TIMESTAMP` constant is interpreted by Matisse in the local time for the client machine. You can also express the constant in Universal Coordinated Time, also known as Greenwich Mean Time, by using the keywords `UTC` or `GMT`.

For instance, if we suppose that the clock for your client machine is set in US Pacific time, which is equivalent to GMT –9, the following constants would actually yield the same internal value:

```
TIMESTAMP '2007-10-01 11:30:00'
TIMESTAMP '2007-10-01 11:30:00' AT LOCAL
TIMESTAMP '2007-10-01 20:30:00' AT GMT
TIMESTAMP '2007-10-01 20:30:00' AT UTC
```

For making your application portable across different time zones, it is strongly recommended that you always store timestamp values in UTC, not in the local time of your machine. Thus, if we suppose that the attribute `lastEntry` contains the timestamp, 2007-10-01 20:30:00, in UTC, the following predicates would evaluate to true:

```
lastEntry =
    TIMESTAMP '2007-10-01 11:30:00' AT LOCAL
lastEntry =
    TIMESTAMP '2007-10-01 20:30:00' AT UTC
```

To get the current timestamp, use the following:

```
CURRENT_TIMESTAMP()
```

This returns the timestamp value in UTC.

Time Interval Constants

A time interval constant is expressed with the following syntax:

```
INTERVAL '[+|-]d hh:mm:ss[.uuuuuu]'
```

where `d` represents the days which can be up to 10 digits, and `hh:mm:ss` respectively represents the hours, minutes, and seconds. An optional fraction of seconds can be specified up to 6 digits.

For instance, if you want to retrieve athlete objects with marathon record less than two hours and ten minutes, you could write a predicate like:

```
marathonRecord < INTERVAL '0 02:10:00.00'
```

Bytes Constants

A bytes constant is a list of unsigned 8-bit integer numbers, where each number is expressed by a pair of hexadecimal digits, has the following syntax:

```
X 'dd...'
```

where `d` represents a hexadecimal digit. Here are some examples:

```
X '000102A0FF'
X '' -- empty bytes
```

List Constants

A list constant is a list of constant values whose types are either integer number, numeric number, real number, boolean, character string, timestamp, date, or time interval. A list constant is expressed with the following syntax:

```
LIST(type)( [constant, ...] )
```

For instance, a list constant with three integer numbers 1, 3, and 5 can be written as follows:

```
LIST(INTEGER)(1, 3, 5)
```

A constant list with two dates can be expressed as follows:

```
LIST(DATE)(DATE '1997-03-10', DATE '1999-11-10')
```

A list of integers with no elements can be expressed as follows:

```
LIST(INTEGER)()
```

Note that all the elements in a constant list need to be of the same type, in particular list elements cannot be NULL.

2.2 What Is an Identifier?

An identifier is a character string possibly enclosed by double quotes (" "). The maximum length of an identifier is 255 characters. The other restrictions are as follows:

- If the identifier is not enclosed by double quotes:
  - It must start with a non-numeric character,
  - It cannot contain separators such as blanks, tabs, carriage returns.
  - The following characters are not allowed:
    `' ` " , . ? ! & ; + - * / % = | ^ ~ ( ) < > [ ] { }`
  - It cannot contain non-displayable characters.
- If the identifier is enclosed by double quotes:
  - It cannot contain carriage returns
  - It cannot contain non-displayable characters
  - A double quote within the identifier is entered by two contiguous double quotes ("").

Matisse SQL is not case sensitive for the identifiers.

2.3 Matisse SQL Reserved Words

Table 2.1 lists the Matisse SQL reserved words that you can use to formulate an SQL request. Matisse SQL keywords are not case sensitive.

Table 2.1 Matisse SQL Reserved Words

ADD	DATE	INTEGER	REF
ALL	DECIMAL	INTERSECT	REFERENCES
ALTER	DECLARE	INTERVAL	RELATIONSHIP
AND	DEFAULT	INTO	RENAME
ANY	DELETE	INVERSE	RESIGNAL
AS	DELETED	IS	RETURN
ASC	DESC	ITERATE	RETURNS
AT	DICTIONARY	JOIN	ROLLBACK
ATTRIBUTE	DIVISION_BY_ZERO	KEY	SELECT
AUDIO	DO	LEAVE	SELECTION
AVG	DOUBLE	LENGTH	SELF
BEGIN	DROP	LIKE	SENSITIVE
BETWEEN	DUPLICATE	LIMIT	SET
BIGINT	ELSE	LIST	SHORT
BLOB	ELSEIF	LOCAL	SIGNAL
BOOLEAN	END	LONG	SMALLINT
BY	ENTRY_POINT	LOOKUP	STATIC
BYTE	ENUM	MAKE_ENTRY	STRING
BYTES	ESCAPE	MESSAGE_TEXT	SUBSCRIBE
CALL	EVENT	METHOD	TABLE
CARDINALITY	EXCEPT	METHODS	TIMESTAMP
CASE	EXIT	MTEXCEPTION	TO
CAST	FALSE	NATURAL	UNFILTERED
CHAR	FILTERED	N	UNIQUE
CHARACTER	FLOAT	NOT	UNKNOWN
CLASS	FOR	NOTIFY	UNSUBSCRIBE

Table 2.1 Matisse SQL Reserved Words *(Continued)*

CLASS_ID	FOREIGN	NULL	UPDATE
CLASS_NAME	FROM	NULL_OBJECT	UPDATED
CLOB	GMT	NUMERIC	UTC
COMMIT	GROUP	NVARCHAR	UTF16
COMPARE	HANDLER	OF	_UTF16
COMPILE	HAVING	OFF	UTF32
COMPILED	IF	OFFSET	UTF7
CONDITION	IMAGE	OID	UTF8
CONNECTION	IN	ON	_UTF8
CONSTRAINT	INDEX	ONLY	VALUES
CONTINUE	INHERIT	OR	VARCHAR
CREATE	INNER	ORDER	VARYING
CURRENT	INOUT	OUT	VERSION
CURRENT_DATE	INSERT	PRECISION	VIDEO
CURRENT_TIMESTAMP	INSERTED	PRIMARY	WAIT
	INSTANCE	READ	WHERE
		READONLY	WORK
		REAL	WRITE
			X

3 Selecting Data

This section explains how to select data. After reading it you should know how to:

- Use the SELECT command
- Name a result with the INTO keyword
- Use predicates in the WHERE clause
- Combine predicates with AND and OR
- Use the NOT keyword to form a negative condition

Here is the syntax for SELECT statement in brief:

```
SELECT [PARALLEL(degree_of_parallelism)]
       [FILTERED | UNFILTERED] [DISTINCT] expression, ...
FROM [ONLY] class, ...
     | selection
     | SELECTION(sel1 {UNION | INTERSECT | EXCEPT} sel2
[WHERE condition]
[GROUP BY attribute, ...]
[HAVING condition]
[ORDER BY attribute [ASC | DESC], ...]
[LIMIT max_number]
[OFFSET start_offset]
[INTO selection]
```

3.1 Using the SELECT Command

You query a Matisse database with the SELECT command. This command returns the objects selected by the selection criteria, or column values specified by the select-list.

This command, in its simplest form, is made up of the SELECT command and a FROM clause. The Select-list part of the SELECT command has the following syntax:

```
SELECT [FILTERED | UNFILTERED] [DISTINCT] {
    *
    | [{class | alias}.][<navigation>.]{<attribute> |
<relationship> | *}
    | OID
    | <expression>
    } [, ...]
```

```
<navigation> ::=
  <relationship>[.([CLASS | ONLY] <class>)]
  [.<relationship>[.([CLASS | ONLY] <class>)] ...]
```

You must specify from where the data will be selected with a `FROM` clause. The `FROM` clause has the following syntax:

```
FROM { [ONLY] class, ...
     | selection
     | SELECTION(sel1 {UNION | INTERSECT | EXCEPT} sel2
     }
```

The following query selects all the objects of the class `movie`:

```
SELECT * FROM movie;
```

Using the ONLY Keyword

You can also use the keyword `ONLY` to select objects of only the class specified in the `FROM` clause, and not any of its subclasses. This is often referred to as the *own instances* of the class, or also the *direct objects* of the class.

For instance, if we suppose that the class `artist` has a subclass `movieDirectors` the following query would select only the objects which are of class `artist` but not `movieDirector` or any other subclass of `artist`:

```
SELECT * FROM ONLY artist;
```

Specifying a SQL Projection

Matisse queries always return a SQL projection. The Matisse C API and language bindings allow you to access the result set and retrieve the values for the columns defined in the Select-list.

The Select-list is a comma separated list of columns which can contain either the symbol *, attributes, relationship, or column expressions.

An *attribute* specification consists of a Matisse attribute name that may be fully qualified with an alias or a class name.

The following statements would return a result set structured into the two columns `firstName` and `lastName`, also referenceable as column 1 and column 2:

```
SELECT firstName, lastName FROM artist;
SELECT artist.firstName, artist.lastName FROM artist;
SELECT a.firstName, a.lastName FROM artist a;
```

A *column expression* specification can be an arithmetic expression or a SQL function including string function, list function, or set function (also called an aggregate function). In the case of a set function, there should be no other column defined in the Select-list. The following statements illustrate the different kinds of column expressions:

```
SELECT title, runningTime/10 FROM movie;
```

```
SELECT m.title, LENGTH(m.directedBy.lastName) FROM movie m;
SELECT AVG(runningTime) FROM movie;
```

List Types in SQL Projection

If an attribute in the Select-list is of list type and the query result is accessed through the Matisse SQL Projection API, then the elements in the list are “exploded” in a similar way a relational join would do. For instance, a box office record with top five receipt numbers would display a result as follows:

```
SELECT week, topReceipts
FROM boxOffice
WHERE week = DATE '2001-01-22';
week             topReceipts
------------     -----------
2001-01-22       16
2001-01-22       11.3
2001-01-22       8.2
2001-01-22       7.6
2001-01-22       7
```

Aliases in SQL Projection

You can also associate a column alias to a projection column, as shown on the following example:

```
SELECT
  AVG(runningTime) AS “avg running time”
FROM
  movie;
```

The column aliases are used in the `HAVING` clause to filter out rows from the result set.

The keyword `AS` is optional and may be omitted.

OID, and Relationship in SQL Projection

The `'SELECT *'` projection includes the attributes, the OID column, and the relationships defined in the class.

For example, the class `movie` has an attribute `title` and a relationship `starring` to `artist` class, class `artist` has an attribute `name`.

```
SELECT * FROM movie;
OID            title             starring
-------------  ---------------   -------------
0x4ff          The Green Mile    0x6e4
0x501          Titanic           0x688
...
```

The OID and relationship columns are of type string and represented by the hexadecimal OID number. Note that the relationship column returns only the first successor object of the relationship for each object, even if the relationship has more than one successor object. This is for the purpose of simplicity.

To get all the starring artists, you may use the following statement.

```
SELECT m.title, a.name AS "Starring Artists"
FROM movie m JOIN artist a ON m.starring = a.OID;
title           Starring Artists
--------------  ---------------------
The Green Mile  Tom Hanks
Titanic         Leonardo DiCaprio
Titanic         Kate Winslet
```

See the next section *3.2, Join Operation* for greater details on join operations.

A simpler version of the above statement would use a navigational query:

```
SELECT m.title, m.starring.name AS "Starring Artists"
FROM movie m;
```

See the *7.4, Navigational Queries*section for more details on navigation queries.

REF() in SQL Projection

The `REF()` built-in can be used in SQL projection to directly access objects from a SQL statement. Unlike `OID` which returns the Object Identifier as a String, `REF()` returns objects and exports them as the C API type `MtOid`. The REF() built-in is heavily used for passing objects from SQL to the object-oriented language bindings. The following table compares OID and REF().

Table 3.1 Comparison of OID and REF()

	OID	REF()
C API type	`MtString`	`MtOid`
Matisse type	`MT_STRING`	`MT_OID`
Primary key	yes	no
Language binding type	`String`	`MtObject`

For example to retrieve all the `Movie` objects released in 2008:

```
SELECT REF(movie)
FROM movie
WHERE releaseYear = 2008;
```

For example to retrieve all the `Actor` objects who star in movies released in 2008:

```
SELECT REF(m.starring)
FROM movie m
WHERE m.releaseYear = 2008;
```

SQL Methods in SQL Projection

The Select-list can include SQL methods.

For example, to list the Directors of the movies released in 1997 assuming the `GetDirectorName()` SQL method is defined on the class `movie`:

```
SELECT m.Title, m.GetDirectorName() AS Director
FROM movie m;
WHERE m.releaseYear = 1997;
Title                  Director
-------------------    -------------
The Horse Whisperer    Robert Redford
Titanic                James Cameron
...
```

The SQL statement below uses both the instance methods `GetAccrualsName()` and `GetAccrualsQuantity()` defined on the `Employee` class and the static method `ListBankName()` defined on the `Bank` class to list the negative accruals for the `Legal` department employees:

```
SELECT
  d.DepartmentName,
  d.employees.EmpId,
  d.employees.LastName,
  d.employees.GetAccrualsName(Bank::ListBankName(FALSE))
AS "Bank Name",
  d.employees.GetAccrualsQuantity(Bank::ListBankName(FALSE
)) AS Total
FROM
  Department d
WHERE
  d.DepartmentName = 'Legal'
HAVING
  Total < 0
ORDER BY
  d.DepartmentName,
  d.employees.LastName;
```

Get a Successor at a Position in a Relationship

You can get a successor object at a specific position in a relationship. For instance, the following SELECT statement returns movie titles with the first starring artist of each movie:

```
SELECT
  m.Name,
  m.Starring(1).FirstName,
  m.Starring(1).LastName,
FROM
  movie m;
```

The syntax to access a successor object at a position in a relationship is

```
relationship_name(position)
```

The first successor object in a relationship is at position 1.

The expression can be used also in `WHERE` clause. For example, the following query statement returns movies whose first starring's last name is Brody:

```
SELECT
  m.Name
FROM
  movie m
WHERE
  m.Starring(1).LastName = 'Brody';
```

Pseudo Attributes

Matisse SQL provides other pseudo attributes besides MtOid, MtClassName and MtClassOid and MtFullClassName defined on the MtObject class as well as MtFullName defined on MtClass, MtIndex, MtEntryPointDictionary, MtAttribute and MtRelationship classes.

MtClassName returns the class name of an object as string. For example,

```
SELECT LastName, MtClassName AS Profession
FROM Artist;

LastName             Profession
-------------------- --------------------
Hanks                Artist
Foster               Artist
Spielberg            MovieDirector
```

MtClassName can be used also in WHERE clause. For example, the next statement returns all the objects whose class name includes 'Corporate':

```
SELECT *
FROM Customer c
WHERE
  c.MtClassName LIKE '%Corporate%';
```

NOTE: Use the IS OF predicate instead of a simple comparison of class name like:

```
SELECT *
FROM Customer c
WHERE c.MtClassName = 'CorporateCustomer';
```

The following predicate executes faster:

```
... WHERE c IS OF (ONLY CorporateCustomer);
```

For more information about the IS OF predicate, refer to *section 3.8, Specifying a Type Predicate with IS OF*.

MtClassOid returns the class of an object in hexadecimal OID format. For example,

```
SELECT LastName, MtClassOid FROM Artist;
LastName            MtClassOid
------------------- -------------------
Hanks               0x25f0
Foster              0x25f0
Spielberg           0x260c
```

The type of MtClassOid is String.

MtFullClassName returns the full qualified class name of an object as string. For example,

```
SELECT LastName, MtFullClassName
FROM Artist;

LastName            MtFullClassName
------------------- -------------------
Hanks               examples.media.Artist
Foster              examples.media.Artist
Spielberg           examples.media.MovieDirector
```

MtFullName returns the full qualified class name of a schema object as string. For example,

```
SELECT MtName, MtFullName FROM MtClass;

MtName    __        MtFullName
------------------- -------------------
Artist              examples.media.Artist
Movie               examples.media.Movie
MovieDirector       examples.media.MovieDirector
```

Pseudo Relationships

Matisse SQL provides several pseudo relationships `MtAllAttributes`, `MtAllRelationships`, `MtAllInverseRelationships`, `MtAllSuperclasses`, `MtAllSubclasses`, `MtAllMethods` defined on `MtClass` to simplify the description class objects with inheritance. Each pseudo relationship navigates through the class hierarchy to the result produced in each individual level of the hierarchy. The `MtAllSubnamespaces` pseudo relationship defined on `MtNamespace` class which returns the combination of the `MtNamespaces` relationship value produced at in each individual level of the namespace path.

`MtAllAttributes` returns the combination of the `MtAttributes` relationship value produced at in each individual level of the class hierarchy. For example,

```
SELECT MtName,MtAttributes.MtName,MtAllAttributes.MtName
FROM MtClass WHERE MtName = 'Manager';
MtName              MtName              MtName
-------------------- -------------------- ----------------
Manager              Expertise            EmpId
Manager             NULL                 LastName
Manager             NULL                 Expertise
```

`MtAllRelationships` returns the combination of the `MtRelationships` relationship value produced at in each individual level of the class hierarchy. For example,

```
SELECT MtName,MtRelationships.MtName,
MtAllRelationships.MtName FROM MtClass WHERE MtName =
'Manager';
MtName              MtName              MtName
-------------------- -------------------- ----------------
Manager            DirectReports        Address
Manager            ManageProjects       Accruals
Manager             NULL                Department
Manager             NULL               DirectReports
Manager             NULL               ManageProjects
```

`MtAllSuperclasses` returns the combination of the `MtSuperclasses` relationship value produced at in each individual level of the class hierarchy. For example,

```
SELECT MtName,MtSuperclasses.MtName,
MtAllSuperclasses.MtName FROM MtClass WHERE MtName =
'Officer';
MtName              MtName              MtName
-------------------- -------------------- ----------------
Officer             Manager             Manager
Officer             NULL                Employee
Officer             NULL                MtObject
```

`MtAllMethods` returns the combination of the `MtMethods` relationship value produced at in each individual level of the class hierarchy. For example,

```
SELECT MtName,MtMethods.MtName,MtAllMethods.MtName FROM
MtClass WHERE MtName = 'Manager';
MtName              MtName              MtName
```

```
-------------------- -------------------- ----------------
Manager              TotalSales           TotalSales
Manager              NULL                 FullName
Manager              NULL                 Expertises
```

3.2 Join Operation

Matisse SQL provides equi-joins among classes using OID as the primary key and relationships as foreign keys. For example, the following statement selects the names of movies along with their director names:

```
SELECT
  m.name, d.lastName, d.firstName
FROM
  movie m,
  movieDirector d
WHERE
  m.directedBy = d.OID;
```

The join condition is expressed using the relationship `directedBy` defined between the two classes. The relationship `directedBy` works as the foreign key and OID works as the primary key. Here are two more examples using a SQL join.

The following statement selects all the movies that have ever passed the $1 million box office record, along with the director's name and box office records.

```
SELECT
  m.name, d.lastName, bx.totalReceipts
FROM
  movie m,
  movieDirector d,
  boxOffice bx
WHERE
  m.directedBy = d.OID AND
  m.boxOfficeRecords = bx.OID AND
  bx.totalReceipts >1000000;
```

You can also join within the same class. Suppose we have the class `person` with a relationship `spouse`. The following statement selects `person` names with spouse's name.

```
SELECT
  p.name, sp.name
FROM
  person p,
  person sp
```

```
WHERE
  p.spouse = sp.OID;
```

Natural Join

If no join condition is provided in the WHERE clause, Matisse SQL tries to find an appropriate one. Since only relationships can work as foreign keys, if there is only one relationship defined between the classes in the FROM clause, Matisse SQL uses the relationship for the JOIN condition.

For example, the boxOfficeRecords relationship is the only one between the movie class and boxOffice class. The following two statements are equivalent:

```
SELECT * FROM movie m, boxOffice bx;

SELECT *
FROM movie m, boxOffice bx
WHERE m.boxOfficeRecords = bx.OID;
```

The following syntax works if there is one and only one relationship between the movie and boxOffice classes, otherwise it returns an error:

```
SELECT * FROM movie NATURAL JOIN boxoffice WHERE ...;
```

The following statement raises an error since there are two relationships directedBy and starring defined between the movie class and movieDirector class. Note that Matisse SQL takes account of inheritance.

```
SELECT * FROM  movie m, movieDirector d; -- error!
```

Conditional Join

The following illustrates the syntax for a conditional join:

```
SELECT *
FROM
  movie m JOIN boxOffice bx
  ON m.boxOfficeRecords = bx.oid
WHERE ...;
```

The ON clause can reference only the joined classes. INNER JOIN may be specified in place of JOIN; the results are the same in either case.

For a three-way conditional join, the syntax is:

```
SELECT *
FROM
  Movie mv JOIN MovieDirector dr
  ON mv.directedBy = dr.oid
  JOIN boxOffice bx ON mv.boxOfficeRecords = bx.oid
WHERE
  dr.lastName = 'Spilberg'
  AND bx.totalReciepts > 10000000;
```

You may use parentheses:

```
SELECT *
FROM
  (Movie mv JOIN MovieDirector dr ON mv.directedBy = dr.oid)
  JOIN boxOffice bx ON mv.boxOfficeRecords = bx.oid
WHERE
  dr.lastName = 'Spilberg'
  AND bx.totalReciepts > 10000000;
```

Sorting the Result

Within a query statement with join operation, you can use as the criteria of sorting (see *section 3.12, Specifying Sort Criteria with ORDER BY*) attributes of the classes specified in the `FROM` clause.

3.3 Using SQL Selections

SQL Selections offer a convenient way to manage a list of objects that are selected from an SQL statement.

Create an SQL Selection

You can create a selection of objects with the keyword **`INTO`**. The keyword `INTO` must be followed with a character string that specifies the name of the new selection result. A `SELECT INTO` statement uses the following syntax:

```
SELECT REF(alias)
FROM classname alias
WHERE ...
INTO selection
```

This selection contains a list of the objects that met the specified criteria. The name for `selection` must be different from that of any of the classes that are accessible in the current context. The following command, for example, selects the objects of the class `movie` and stores them in a new selection called `mvAction`:

```
SELECT REF(m) FROM movie m INTO mvAction;
```

Note that no projection is printed in the Enterprise Manager or in the `mt_sql` utility when a SELECT statement has an `INTO` clause to generate a selection.

Select from SQL Selections

After executing the above command, you can select the objects from the `mvAction` selection, as shown below:

```
SELECT * FROM mvAction;
```

The name for selection can be the same as any selection previously used in the current transaction and still accessible, in which case the selection will be overwritten with a new list of objects.

For example, you can narrow down the `mvAction` selection with a `WHERE` clause, as shown below:

```
SELECT REF(m) FROM mvAction m WHERE ... INTO mvAction;
```

You can execute a SELECT statement from the result of a set operation on selections like `UNION`, `INTERSECT`, or `EXCEPT`. For example, you create two selections for Movie objects:

```
SELECT REF(m) FROM Movie m WHERE ... INTO movies1;
SELECT REF(m) FROM Movie m WHERE ... INTO movies2;
```

then, you can select movies from the above two selections with additional criteria using the SELECTION syntax. For example,

```
SELECT m.Title,
FROM SELECTION(movies1 UNION movies2) m
WHERE m.Name LIKE ‘M%’;
```

For the intersection of two selections, use INTERSECT:

```
... FROM SELECTION(movies1 INTERSECT movies2)
```

For the difference of two selections, use EXCEPT:

```
... FROM SELECTION(movies1 EXCEPT movies2)
```

A nested set operation on selections is allowed. For instance,

```
... FROM SELECTION(SELECTION(movies1 INTERSECT movies2)
                       INTERSECT movies3)
```

Selection Class

When the classes of selections are identical, like the above examples using movies, it is obvious from which class you are selecting objects. However, when the classes of selections are different, you need to specify the class from which you are selecting.

For example, suppose you have three classes `Employee`, `HourlyEmployee`, and `SalariedEmployee`, where the last two classes are inheriting from class `Employee`. You created two selections `hourly` and `salaried` from `HourlyEmployee` and `SalariedEmployee`, respectively. Then, you select from the union of the two selections:

```
SELECT FirstName, LastName
FROM SELECTION(hourly UNION salaried) AS Employee
WHERE ...
```

If you do not specify the class alias, you cannot access the properties defined in class Employee. If you specify more specialized class then the common super classes, you will get an error.

Delete a Selection

SQL Selections created with the `INTO` keyword as shown in the this section must be deleted using a `DROP SELECTION` statement when they are no longer needed.

The syntax for `DROP SELECTION` is as follows:

```
DROP SELECTION selection
```

For instance, an application may run the following queries:

```
SELECT REF(m) FROM movie m INTO mvAction;
... [other queries using the selection] ...
DROP SELECTION mvAction;
```

3.4 Specifying a Search Criteria with WHERE

A search criterion can be defined in the `WHERE` clause as a combination of predicates. During execution the predicates are evaluated on the objects specified in the `FROM` clause. Each predicate evaluates to one of the following three values:

```
TRUE
FALSE
UNKNOWN
```

Each object for which the combination of the predicates evaluated `TRUE` is added to the selection result. Objects for which the evaluation returned `FALSE` or `UNKNOWN` are not added to the selection result.

The following example shows how to select objects of the class `movie` with a running time longer or equal 90 minutes:

```
SELECT REF(m)
FROM movie m
WHERE (runningTime >= 90)
INTO mvAction;
```

Note that the predicate `(runningTime >= 90)` compares the value of the numeric attribute `runningTime` to the constant `90`. Only those objects of class `movie` that qualify for this predicate will be added in the selection result `mvAction`.

In the `SELECT` request shown above, it is obvious that the objects for which the comparison is `FALSE` are those whose `runningTime` is less than 90 minutes. The objects for which the comparison is `UNKNOWN` are those for which the `runningTime` is a null value or is not a numeric type.

3.5 Using Attributes in Expressions

You can specify attribute expressions either in the Select-list to define an SQL projection, or as part of an evaluation predicate in the `WHERE` clause.

A predicate where one value is compared to another has the following syntax:

```
expression1 comparison_operator expression2
```

A predicate expression can contain any of the attributes of the class specified in the `FROM` clause. The set of possible types associated with the attribute is the set of types associated with the descriptor for the attribute in the database schema.

Specifying an Attribute in a WHERE Clause

When you specify an attribute expression in the `WHERE` clause, you can specify the attribute by itself or preceded by a class name or an alias. In any case, the attribute that you specify must belong to the class specified in the `FROM` clause.

Here is the syntax for specifying an attribute:

```
[ ( class | alias ).] attribute
```

In the example below we specify the attribute `runningTime` without a class or an alias qualifier:

```
SELECT *
FROM movie
WHERE runningTime = 90;
```

In the example below we specify two attributes preceded by the class name qualifier:

```
SELECT *
FROM movie
WHERE
  movie.title LIKE 'Rocky%'
  AND movie.runningTime > 90;
```

The same query using an alias qualifier instead of the class name is shown below:

```
SELECT *
FROM movie AS m
WHERE
  m.title LIKE 'Rocky%'
  AND m.runningTime > 90;
```

3.6 Combining Predicates with AND and OR

You can combine two or more predicates with the `AND` and `OR` logical operators. Predicates linked together by these logical operators have the following syntax:

```
predicate1 logical_operator predicate2
```

When connected by an AND operator, both predicates must evaluate to true for the AND to evaluate to true. When connected by an OR operator, only one of the predicates needs to evaluate to true for the OR to evaluate to true.

The following example might help illustrate compound predicates. If you want to select movies that have a running time greater than 90 minutes and a title starting with ‘Rocky’.

A request like this would have the following syntax:

```
SELECT *
FROM movie
WHERE
  runningTime > 90
  AND title LIKE 'Rocky%';
```

The result of the evaluation of the conjunctions (AND) and unions (OR) of predicates are defined on truth tables. Table 3.2 is the truth table for the AND operator.

Table 3.2 AND Operator Truth Table

Predicate 1	Predicate 2	Result
True	True	True
True	False	False
False	True/False	False
Unknown	True/False/Unknown	Unknown

Table 3.3 is the truth table for the OR operator.

Table 3.3 OR Operator Truth Table

Predicate 1	Predicate 2	Result
True	True/False/Unknown	True
False	False	False
Unknown	False/Unknown	Unknown

Precedence of Evaluation of AND and OR

The subpredicates expressed within parentheses are evaluated in priority. For operations at the same level, AND operators are applied before OR operators.

When a predicate does not have parentheses, a predicate is then interpreted from left to right. The predicate

```
A AND B AND C
```

for example, is equivalent to the following predicate:

```
(A AND B) AND C
```

Matisse SQL implements the classic laws of commutativity and distributivity for the AND and OR operators, as shown in *Table 3.4*.

Table 3.4 Equivalent Logical Expressions

Expression	Equivalent Expression
A AND B	B AND A
A OR B	B OR A
A AND (B OR C)	(A AND B) OR (A AND C)
A OR (B AND C)	(A OR B) AND (A OR C)

A SELECT statement that selects objects of the class movie where the runningTime is greater than 120 minutes or less than 90 minutes and whose title starts with 'Rocky' would look like:

```
SELECT *
FROM movie
WHERE
  title LIKE 'Rocky%'
  AND (runningtime < 90 OR runningTime > 120);
```

Note that in accordance with the law of distributivity described above, the following request is equivalent:

```
SELECT *
FROM movie
WHERE
  (title LIKE 'Rocky%' AND runningTime < 90)
  OR
  (title LIKE 'Rocky%' AND runningTime > 120);
```

3.7 Specifying a Negative Condition with NOT

You can use the NOT keyword to evaluate the opposite or negation of a predicate.

For example, to select objects of the class movie that do not have a title starting with 'Rocky', you could write the following statement:

```
SELECT *
FROM movie
WHERE
  NOT title LIKE 'Rocky%';
```

A SELECT statement that selects objects of the class movie where the runningTime is greater than 120 minutes or less than 90 minutes and whose title starts with 'Rocky' would look like:

```
SELECT *
FROM movie
WHERE
  title LIKE 'Rocky%'
  AND NOT runningtime BETWEEN 90 AND 120;
```

3.8 Specifying a Type Predicate with IS OF

A type predicate tests object instances based on their classes. The syntax is as follows:

```
expression IS [NOT] OF ([ONLY] classname [, ...])
```

where *expression*, representing an object, is a class name or alias name specified in the FROM clause, or relationship navigations. The result of the predicate is true if
i) the actual class of an object, *expression*, is *classname* or one of the subclasses of *classname*, or
ii) the actual class of an object is *classname* if the optional ONLY precedes *classname*,
for at least one of the classes specified by *classname*.

If *expression* is NULL, the result of the predicate is unknown.

For example, the next SELECT statement selects employees using different conditions for different type of employee:

```
SELECT *
FROM Employee e
WHERE
  (e IS OF (ONLY Employee) AND salary > 40000)
  OR
  (e IS OF (Manager, Officer) AND salary > 50000);
```

When *expression* contains relationship navigations, the predicate executes the type test for each successor object of the relationship. If at least one of the successor object satisfies the type test, the result of the predicate is true.

For example, the following statement selects movies which has any starring movie director:

```
SELECT *
FROM Movie m
WHERE
  m.starring IS OF (MovieDirector);
```

Note that if the relationship `starring` has no successor object, the type predicate evaluates to unknown since `m.starring` is NULL.

3.9 Specifying UNFILTERED

The `UNFILTERED` query hint forces a direct `SELECT` statement to build the full SQL projection on the server-side the same way a block statement or a SQL Method would do it, thus eliminating the need for returning objects to the client workspace.

```
SELECT UNFILTERED
   d.DepartmentName,
   d.employees.EmpId,
   d.employees.Salary
FROM
   Department d
ORDER BY
   d.DepartmentName;
```

The SQL statement above is equivalent to the same SELECT statement executed in a block statement:

```
BEGIN
  SELECT
    d.DepartmentName,
    d.employees.EmpId,
    d.employees.Salary
  FROM
    Department d
  ORDER BY
    d.DepartmentName;
END;
```

3.10 Navigation Filtering with FILTERED

The `SELECT FILTERED` statement applies the relationship navigation filters in the `WHERE` clause to the relationship navigation in the Select-list. The SQL projection produced is equivalent to the SQL projection of a SQL relational equi-join.

The SELECT statements below shows the effect of `FILTERED` on the SQL projection result. The first statement without `FILTERED` returns only the Department matching all the conditions expressed in the `WHERE` clause and in these departments all the employees are returned.

```
SELECT
  d.DepartmentName,
  d.employees.EmpId,
  d.employees.Salary
FROM
```

```
  Department d
WHERE
  d.employees.Salary = 180000;
DepartmentName   EmpId          Salary
---------------  -------------  -----------------
Finance          48             135000
Finance          47             145000
Finance          46             160000
Finance          45             145000
Finance          3              180000
Finance          2              200000
Finance          1              249000
Finance          440            90000
Finance          439            45000
...
```

The second statement, which includes `FILTERED`, returns only the rows matching all the conditions expressed in the `WHERE` clause that is only 2 rows:

```
SELECT FILTERED
   d.DepartmentName,
   d.employees.EmpId,
   d.employees.Salary
FROM
   Department d
WHERE
   d.employees.Salary = 180000;
DepartmentName   EmpId          Salary
---------------  -------------  -----------------
Finance         3               180000
Finance         19              180000
```

The result of the `SELECT FILTERED` statement is equivalent to the result of a `SELECT` statement with a `HAVING` clause that includes the navigation predicates defined in the `WHERE` clause of the `SELECT FILTERED`.

```
SELECT
  d.DepartmentName,
  d.employees.EmpId,
  d.employees.Salary
FROM
  Department d
HAVING
  Salary = 180000;
DepartmentName   EmpId          Salary
---------------  -------------  -----------------
Finance         3               180000
```

```
Finance         19              180000
```

Matching Predicates

The combination of navigation predicates expressed in the WHERE clause are applied to the best matching columns in the Select-list. For example the following statement that combines AND and NOT predicates, lists all the Manager expert in Design and Full Time employee, working on a project with project members not being Contractor:

```
SELECT FILTERED
   d.DepartmentName,
   d.employees.Manager.EmpId,
   d.employees.Manager.ManageProjects.ProjectID,
   d.employees.Manager.ManageProjects.Members.EmpId
FROM
   Department d
WHERE
   'Design' IN d.employees.Manager.Expertise
   AND d.employees.Manager.Contract = 'Full Time'
   AND NOT
d.employees.Manager.ManageProjects.Members.Contract =
'Contractor'
ORDER BY
   d.DepartmentName,
   d.employees.Manager.EmpId;
```

First note that in the Select-list the employees relationship includes the Manager class filter. The first 2 predicates match with the second column and the last predicate matches with the last column.

Matching Predicates with Composition

The navigation predicates ending with a Composition (“Part of”) relationship are applied to the column that matches the navigation path parent of the Composition relationship. For instance, the statement below lists all the Manager expert in Design and living in Sevilla, working on a project with project members living in Charleroi or Stuttgart or Strasbourg:

```
SELECT FILTERED
   d.DepartmentName,
   d.employees.Manager.EmpId,
   d.employees.Manager.ManageProjects.ProjectID,
   d.employees.Manager.ManageProjects.Members.EmpId
FROM
   Department d
WHERE
   'Design' IN d.employees.Manager.Expertise
   AND d.employees.Manager.Address.City = 'Sevilla'
```

```
    AND
d.employees.Manager.ManageProjects.Members.Address.City IN
( 'Charleroi', 'Stuttgart', 'Strasbourg')
ORDER BY
    d.DepartmentName,
    d.employees.Manager.EmpId;
```

The first 2 predicates match with the second column and the last predicate matches with the last column.

Unmatching Predicates

When navigation predicates are deeper than the column navigation paths, they are applied to the deepest matching column of the Select-list. The next statement is similar to the one defined in the section above, except that it lists only the Employee Identifier for all the `Manager` expert in `Design` living in `Sevilla`, working on a project with project members living in `Charleroi` or `Stuttgart` or `Strasbourg`:

```
SELECT FILTERED
    d.DepartmentName,
    d.employees.Manager.EmpId
FROM
    Department d
WHERE
    'Design' IN d.employees.Manager.Expertise
    AND d.employees.Manager.Address.City = 'Sevilla'
    AND
d.employees.Manager.ManageProjects.Members.Address.City IN
( 'Charleroi', 'Stuttgart', 'Strasbourg')
ORDER BY
    d.DepartmentName,
    d.employees.Manager.EmpId;
```

In this statement, all the predicates match with the second column.

Filtering and Reordering Relationship with REF()

The `REF()` built-in can be used in a SQL projection to directly access objects from a SQL statement. `REF()` combined with `FILTERED` and `ORDER BY` makes a powerful mean to filter and to reorder relationships. The following SQL statement retrieves all the Employees in each department an reorder the Employee objects by salary in descending order:

```
SELECT FILTERED
   REF(d),
   REF(d.Employees)
FROM
   Department d
ORDER BY
   d.DepartmentName,
   d.Employees.Salary DESC;
```

The next statement filters the Employee objects based on their class. It retrieves all the Managers (excluding subclasses) from the `Finance` department and reorder the objects by employee identifier:

```
SELECT FILTERED
  REF(d),
  REF(d.Employees.(ONLY Manager))
FROM
  Department d
WHERE
  d.DepartmentName = 'Finance';
ORDER BY
  d.DepartmentName,
  d.Employees.EmpId;
```

The last example filters the Employee objects based on some property values. It retrieves all the Managers and their Direct Reports who are `Contractor` and expert in `Design` and reorder the Direct Reports Employee objects by their Last Name:

```
SELECT FILTERED
  REF(m),
  REF(m.DirectReports.Employee)
FROM
  Manager m
WHERE
  m.DirectReports.Contract = 'Contractor'
  AND 'Design' IN m.DirectReports.Expertise
ORDER BY
  m.EmpId,
  m.DirectReports.LastName;
```

Relationship COUNT

The `WHERE` clause navigation predicates do not apply to the Relationship Count built-in (`COUNT`). For example, the statement below counts the number of employees in each department where there is at least one Executive and not the number of Executive:

```
SELECT FILTERED
   d.DepartmentName,
   count(d.employees) AS "Exec Count"
FROM
   Department d
WHERE
   d.employees IS OF (ONLY Officer,Executive)
ORDER BY
   d.DepartmentName;
```

The Relationship Count built-in (COUNT) supports only explicit class filtering. The following statement returns the Executive head count in each department:

```
SELECT FILTERED
  d.DepartmentName,
  COUNT(d.employees.Officer) +
  COUNT(d.employees.Executive) AS "Exec Count"
FROM
  Department d
ORDER BY
  d.DepartmentName;
```

The Object Count built-in (COUNT) does support navigation filtering. The next statement also returns the number of Executives in each department where there is at least one Executive:

```
SELECT FILTERED
   d.DepartmentName,
   count(d.employees.*) AS "Exec Count"
FROM
   Department d
WHERE
   d.employees IS OF (ONLY Officer,Executive)
GROUP BY
   d.DepartmentName
ORDER BY
   d.DepartmentName;
```

3.11 Getting DISTINCT Values

When you want to get only one copy for each set of duplicate rows, use the DISTINCT keyword in the select-list. For example, the following statement lists all the kinds of ratings for each category:

```
SELECT DISTINCT category, rating FROM movie;
```

In the current release, **DISTINCT *** applies only to the properties defined in the ORDER BY clause and allows you to select only scalar values excluding list types and multimedia types. Note that retrieving the DISTINCT values in a list type attribute can easily be done with a SQL Method. The following **DISTINCT *** statement requires to specify each DISTINCT property in the ORDER BY clause.

```
SELECT DISTINCT * FROM movie ORDER BY category, rating;
```

To retrieve distinct values in navigational queries, you need to specify the navigation path for each DISTINCT property in the ORDER BY clause as follows:

```
SELECT DISTINCT
```

```
  d.DepartmentName,
  CONCAT('"',CONCAT(d.employees.Contract, '"')) AS
  Contract,
  CAST((d.employees.Salary / 12) AS INT) AS Monthly
FROM
  Department d
ORDER BY
  d.DepartmentName,
  d.employees.Contract,
  d.employees.Salary DESC;
```

3.12 Specifying Sort Criteria with ORDER BY

You can use an `ORDER BY` clause to sort the objects according to the values of some of the attributes. You can specify the order to be ascending or descending for each attribute in the `ORDER BY` clause. By default, the order is ascending.

The syntax is as follows:

```
ORDER BY criteria
```

Where *`criteria`* is a list of comma-separated criteria with each criterion having the following syntax:

```
{ [{class | alias }.][navigation.]attribute [ASC | DESC] }
```

With navigation such as:

```
navigation ::=
  relationship[.({CLASS | ONLY} successor_class)]
  [.relationship[.({CLASS | ONLY} successor_class)] ...]
```

Note that criteria attribute cannot be of list types nor multimedia types, e.g., `LIST(INTEGER)`, `IMAGE`, or `VIDEO`.

For instance, to select the movies by `title` ascending and `runningTime` descending, with a running time higher than 90 minutes, you would write the following statement:

```
SELECT *
FROM movie
WHERE runningTime > 90
ORDER BY title ASC, runningTime DESC;
```

Note that the ascending or descending specification is "sticky," it propagates to the next criteria unless otherwise specified. For instance the following statement will sort the objects on *both* `title` *and* `runningTime` descending, as the `DESC` propagates to the right.

```
SELECT *
FROM movie
ORDER BY title DESC, runningTime;
```

To sort the objects on `runningTime` ascending, you need to specify it explicitly:

```
SELECT *
FROM movie
ORDER BY title DESC, runningTime ASC;
```

To sort the objects in navigational queries, you need to specify the navigation path in the `ORDER BY` clause as follows:

```
SELECT m.title, m.starring.lastName AS Starring
FROM movie m
ORDER BY m.title, m.starring.lastName DESC;
```

Note that within a query statement containing a `JOIN` operation, you may use attributes of the classes specified in the `FROM` clause as sort criteria (that is, as arguments to the `ORDER BY` clause).

3.13 Filtering with HAVING

The `HAVING` clause is a mean to filter out rows from a projection of a navigational query. All predicates in the `HAVING` clause must use aliases from the Select-list.

For instance, the following statement filter out the rows in the result set that does not match the `HAVING` predicates:

```
SELECT
   d.DepartmentName,
   d.employees.EmpId,
   d.employees.Salary AS Salary
FROM
   Department d
HAVING
   Salary = 180000
ORDER BY
   d.DepartmentName;
DepartmentName   EmpId          Salary
---------------  -------------  -----------------
Finance         3               180000
Finance         19              180000
```

The `SELECT / HAVING` statement above is equivalent to a `SELECT FILTERED` with a `WHERE` clause statement which filters on the employee's salary:

```
SELECT FILTERED
   d.DepartmentName,
   d.employees.EmpId,
   d.employees.Salary AS Salary
FROM
   Department d
WHERE
   d.employees.Salary = 180000
ORDER BY
   d.DepartmentName;
DepartmentName   EmpId          Salary
---------------  -------------  -----------------
Finance         3              180000
Finance         19             180000
```

Filering List Type Values

The HAVING clause is the only mean to filter out rows from object properties, variables or SQL methods returning a LIST of values. For instance, to select the Directors expert in `'Organization'` and showing only this expertise, you would write the following statement:

```
SELECT FILTERED
   d.DepartmentName,
   d.employees.Director.EmpId,
   d.employees.Director.LastName,
   d.employees.Director.Expertise AS Expertise
FROM
   Department d
WHERE
   d.DepartmentName = 'Engineering'
   AND 'Organization' IN d.employees.Director.Expertise
HAVING
   Expertise = 'Organization'
ORDER BY
   d.DepartmentName,
   d.employees.Director.EmpId;
```

To select the Directors expert in `'Organization'` or `'Management'` resulting from the execution of a SQL Method and showing only these skills, you would write the following statement:

```
SELECT
   d.DepartmentName,
   d.employees.Director.EmpId,
   d.employees.Director.LastName,
   d.employees.Director.getRankedExpertises(7) AS Skills
```

```
FROM
   Department d
WHERE
   d.DepartmentName = 'Engineering'
HAVING
   Skills = 'Development' OR Skills = 'Management'
ORDER BY
   d.DepartmentName,
   d.employees.EmpId;
```

Aggregate Values from SQL Methods

Aggregate values could also result from the execution of SQL methods. A SQL methods are a convenient and powerful way to compute application-specific aggregate values. For instance, the following SQL statement uses the `GetDirectReportsTotalSalaries()` method to calculate the total budget of a manager:

```
SELECT FILTERED
  d.DepartmentName,
  d.employees.Manager.Class_Name AS Position,
  d.employees.Manager.EmpId,
  d.employees.Manager.LastName,
  COUNT(d.employees.Manager.DirectReports) AS "Head Count",
  d.employees.Manager.GetDirectReportsTotalSalaries() AS
  Budget
FROM
  Department d
WHERE
  d.DepartmentName = 'Legal'
HAVING
  (Budget / "Head Count") >= 80000
ORDER BY
  d.DepartmentName,
  d.employees.Manager.LastName;
```

Note that this statement could also be expressed with a `GROUP BY` clause if the `GetDirectReportsTotalSalaries()` method was just computing the sum of the direct report employees salary as follows:

```
SELECT FILTERED
  d.DepartmentName,
  d.employees.Manager.Class_Name AS Position,
  d.employees.Manager.EmpId,
  d.employees.Manager.LastName,
  COUNT(d.employees.Manager.DirectReports.*) AS
  "Head Count",
  SUM(d.employees.Manager.DirectReports.Salary) AS
```

```
    Budget
  FROM
    Department d
  WHERE
    d.DepartmentName = 'Legal'
  GROUP BY
    d.DepartmentName,
    d.employees.Manager.LastName,
    d.employees.Manager.Class_Name,
    d.employees.Manager.EmpId
  HAVING
    (Budget / "Head Count") >= 80000
  ORDER BY
    d.DepartmentName,
    d.employees.Manager.LastName;
```

Another example of a SQL statement that could use SQL methods to calculate the employee's accruals:

```
  SELECT UNFILTERED
    d.DepartmentName,
    d.employees.EmpId,
    d.employees.LastName,
    d.employees.GetAccrualsName(Bank::ListBankName(FALSE))
    AS "Bank Name",
    d.employees.GetAccrualsQuantity(Bank::ListBankName(FALSE
  )) AS Total
  FROM
    Department d
  WHERE
    d.DepartmentName = 'Legal'
  HAVING
    Total < 0
  ORDER BY
    d.DepartmentName,
    d.employees.LastName;
```

NOTE: When a HAVING clause is used without GROUP BY, the entire objects resulting from the WHERE clause is treated as a single group. Then, the statement's Select-list can contain only set functions, since nothing is specified in GROUP BY clause.

3.14 Grouping with GROUP BY

When a GROUP BY clause is used with a SELECT statement, the GROUP BY clause groups the selected objects based on the values of attributes specified by GROUP BY clause, and returns a single row as summary information for each grouped objects.

NOTE: All NULL values from grouping attributes are considered equal.

The syntax is:

```
SELECT ...
WHERE ...
GROUP BY property [, ...]
[HAVING <search condition>]
[ORDER BY property [ASC | DESC] [, ...]]
```

Where *property* is a list of comma-separated properties with each property having the following syntax:

```
{ [{class | alias }.][navigation.]attribute }
```

With navigation such as:

```
navigation ::=
  relationship[.({CLASS | ONLY} successor_class)]
  [.relationship[.({CLASS | ONLY} successor_class)] ...]
```

A GROUP BY clause can have up to 16 properties as its grouping criteria. Note that grouping attribute cannot be of list types nor multimedia types, e.g., `LIST(INTEGER)`, `IMAGE`, or `VIDEO`.

A simple example is to group movies based on their categories and return the average running time for each group:

```
SELECT category, AVG (runningTime)
FROM Movie
GROUP BY category;
category          avg
----------------- ------------
           Action        108.5
            Drama        125.1
```

When a GROUP BY clause is used, the Select-list can reference:

a. attributes specified in the GROUP BY clause, or
b. any attribute that is used as parameter for set function.

And also, the ORDER BY clause can reference only attributes specified in the GROUP BY clause.

For example, the next statement is valid:

```
SELECT
  CONCAT ('Category: ', category),
  AVG (runningTime)
FROM Movie
GROUP BY category;
```

while the following is invalid:

```
SELECT category, title
FROM Movie
GROUP BY category;    -- Error!!
```

because `title` is neither a grouping attribute nor used as parameter for a set function.

Grouping by Class

MtClassName or MtClassOid can be used to group objects by their class. For example,

```
SELECT
  MtClassName,
  AVG(salary)
FROM Employee
GROUP BY
  MtClassName;
MtClassName           avg
-------------------- --------------
Employee                       23504.23
Manager                        32119.13
```

In the example, the objects in the group of Employee consist of only direct instances of class `Employee`, excluding objects of class `Manager`, which is a subclass of `Employee`.

Grouping with Navigation

You can group by objects in navigational queries by specifying the navigation path in the `GROUP BY` clause. For example, the following statement lists the number of contracts per contract type, for each employee position in each department:

```
SELECT
   d.DepartmentName,
   d.employees.Class_Name AS Position,
   d.employees.Contract,
   count(d.employees.*)
FROM
   Department d
GROUP BY
   d.DepartmentName,
   d.employees.Class_Name,
```

```
      d.employees.Contract
   ORDER BY
      d.DepartmentName,
      d.employees.Class_Name DESC,
      d.employees.Contract ASC;
```

Grouping by Composition

You can group by objects based on the values of “part-of” attributes via a Composition relationship. For example, the following statement lists the number of employee per city:

```
SELECT
   e.Address.City,
   COUNT(*)
FROM
   Employee e
GROUP BY
   e.Address.City
ORDER BY
   e.Address.City DESC;
```

A Composition (“Part of”) relationship in a GROUP BY clause is a relationship with a maximum cardinality of 1. For example, the following statement lists the total employees salary for each department:

```
SELECT
   e.Department.DepartmentName,
   SUM(e.Salary) AS "Total Salary"
FROM
   Employee e
GROUP BY
   e.Department.DepartmentName
ORDER BY
   e.Department.DepartmentName DESC;
```

While the SQL statement above is correct, the SQL Statement below is equivalent to the previous one, but does not require a `GROUP BY` clause since the Department Names are unique:

```
SELECT
   d.DepartmentName,
   SUM(d.employees.Salary) AS "Total Salary"
FROM
   Department d
ORDER BY
   d.DepartmentName DESC;
```

For example, the following statement combines Composition with navigation:

```
SELECT
```

```
    d.DepartmentName,
    d.employees.Address.City,
    SUM(d.employees.Salary)
FROM
    Department d
GROUP BY
    d.DepartmentName,
    d.employees.Address.City
ORDER BY
    d.DepartmentName DESC,
    d.employees.Address.City;
```

3.15 Filtering with HAVING in GROUP BY

The `HAVING` clause can restrict the groups of the selected objects to those groups for which the *search condition* is true. All predicates in the `HAVING` clause must use aliases from the Select-list.

For example, the following statement selects movie categories in which average running time is more than two hours:

```
SELECT category, AVG (runningTime) AS AvgTime
FROM Movie
GROUP BY category
HAVING AvgTime > 120;

category          AvgTime
----------------- ------------
            Drama        125.1
```

The HAVING clause can reference only aliases from the Select-list.

GROUP BY / HAVING with Navigation

The following statement shows how you can group filtered objects in a navigational query by class, property and composition and restrict the resulting groups with a fairly complex combination of predicates:

```
SELECT FILTERED
    Location,
    d.Employees.Contract,
    d.Employees.Class_Name AS Position,
    d.Employees.EmpId,
    d.Employees.LastName,
    d.Employees.Accruals.Bank.BankName AS BankName,
    SUM(d.Employees.Accruals.Quantity) AS Total
FROM
    Department d
```

```
WHERE
   (Location = 'Seattle'
    AND d.Employees.Contract = 'Part Time')
   OR
   (Location = 'Strasbourg'
    AND d.Employees.Contract = 'Contractor')
GROUP BY
   d.Location,
   d.Employees.Contract,
   d.Employees.Class_Name,
   d.Employees.LastName,
   d.Employees.EmpId,
   d.Employees.Accruals.Bank.BankName
HAVING
  (BankName = 'Worked Hours' AND Total > 8.00)
  OR
  (BankName IN LIST(STRING)('Overtime', 'Holiday Overtime')
   AND Total < 0);
```

The next statement shows how to compute complex aggregate values for each group and how in the HAVING clause you can define predicates that compare columns:

```
SELECT FILTERED
   d.DepartmentName,
   d.employees.Class_Name AS Position,
   d.employees.Contract,
   CAST((MIN(d.employees.Salary)/12) AS INT) AS MinSalary,
   CAST((MAX(d.employees.Salary)/12) AS INT) AS MaxSalary,
   COUNT(d.employees.*) AS HeadCount,
   CAST((MIN(d.employees.Salary)/12) AS INT) *
COUNT(d.employees.*) AS MinTotal,
   CAST((MAX(d.employees.Salary)/12) AS INT) *
COUNT(d.employees.*) AS MaxTotal
FROM
   Department d
WHERE
   d.DepartmentName = 'Sales' AND
   ( d.employees.MonthlySalary() between 4000 and 5000 OR
     d.employees.MonthlySalary() between 9000 and 11000 )
GROUP BY
   d.DepartmentName,
   d.employees.Class_Name,
   d.employees.Contract
HAVING
  (MinSalary = MaxSalary AND HeadCount > 1)
```

```
OR
(MinTotal * 2 > MaxTotal);
```

3.16 LIMIT and OFFSET

The LIMIT and OFFSET clauses allow you to retrieve a portion of objects that are selected by a where-clause.

```
LIMIT {count | ALL}
OFFSET start_offset
```

If a `LIMIT` clause is specified, no more than *`count`* objects are returned. If *`count`* is 0, it is the same as omitting the `LIMIT` clause. `LIMIT ALL` is the same as omitting the `LIMIT` clause.

OFFSET specifies the number (*`start_offset`*) of objects to skip from the beginning of the selected objects.OFFSET 0 is the same as omitting the OFFSET clause.

If both LIMIT and OFFSET are specified, *`start_offset`* number of objects are skipped before counting the LIMIT objects.

When you use LIMIT and/or OFFSET, it is always a good idea to use them with an ORDER BY clause that transforms the result into a unique order. Otherwise, you will get unpredictable result.

For example, the next query returns movies from #21 to #30 when ordered by movie's title:

```
SELECT * FROM Movie ORDER BY Title LIMIT 10 OFFSET 20;
```

The next statement selects the Movie object that has the largest sales in a stored method:

```
BEGIN
  DECLARE mvObj Movie;
  DECLARE vLimit Integer DEFAULT 1;
  SELECT REF(m) INTO mvObj FROM Movie m
   ORDER BY TotalSales DESC
   LIMIT vLimit;
  ...
END;
```

The next statement limits the number of iteration of the FOR statement to no more than 10 in a stored method:

```
BEGIN
  FOR obj AS SELECT REF(c) FROM Movie c LIMIT 10 DO
    ....
  END FOR;
```

```
  ...
END;
```

3.17 Subqueries

Although the current release of Matisse does not support subqueries, the same queries can be achieved by using Block statements or SQL methods. With SQL methods, queries become more readable and extensible, and they can be polymorphic.

Here are a few examples demonstrating how you can rewrite subqueries with Block statements and SQL methods.

Subquery for Comparison

A very common use of subquery is a query that returns a scalar value which is used for value comparison. For example,

```
SELECT * FROM Class1 c1
 WHERE c1.attr1 > (SELECT AVG(c2.attr2) FROM Class2 c2);
```

the above query can be rewritten using the SQL method defined below:

```
SELECT * FROM Class1 c1
 WHERE c1.attr1 > Class2::AvgOfAttr2();

CREATE STATIC METHOD AvgOfAttr2()
RETURNS DOUBLE
FOR Class2
BEGIN
  DECLARE avg DOUBLE;
  SELECT AVG(attr2) INTO avg FROM Class2;
  RETURN avg;
END;
```

the above query can be rewritten using the SQL Block statement defined below:

```
BEGIN
  DECLARE avg DOUBLE;
  SELECT AVG(attr2) INTO avg FROM Class2;
  SELECT * FROM Class1 c1 WHERE c1.attr1 > avg;
END;
```

Subquery used with IN

Subqueries used with IN predicate can be rewritten using an SQL method that returns a value of list type. For example,

```
SELECT * FROM Class1 c1
 WHERE c1.attr1 IN
    (SELECT c2.attr2 FROM Class2 c2
```

```
        WHERE c2.attr22 = c1.attr11);
```

The above query can be rewritten as following:

```
SELECT * FROM Class1 c1
 WHERE c1.attr1 IN Class2::method2(c1.attr11);
```

where the SQL method is defined below:

```
CREATE STATIC METHOD method2(arg1 INTEGER)
RETURNS LIST(INTEGER)
FOR Class2
BEGIN
  DECLARE res LIST(INTEGER) DEFAULT LIST(INTEGER)();
  FOR obj AS SELECT REF(c) FROM Class2 c WHERE attr22 = arg1
DO
    ADD(res, obj.attr2);
  END FOR;
  RETURN res;
END;
```

This example demonstrates that correlated subqueries can be substituted by SQL methods as well.

Subquery with EXISTS

The existence test with a subquery can also be rewritten using an SQL method that returns true or false based on the number of selected rows. For example,

```
SELECT * FROM Class1 c1
 WHERE EXISTS
     (SELECT * FROM Class2 c2 WHERE c2.attr2 > c1.attr1);
```

The above statement can be rewritten as following:

```
SELECT * FROM Class1 c1
 WHERE Class2::ObjsExist(c1.attr1) = TRUE;
```

where the SQL method is defined as follows:

```
CREATE STATIC METHOD ObjsExist(arg1 INTEGER)
RETURNS BOOLEAN
FOR Class2
BEGIN
  DECLARE cnt INTEGER;
  SELECT COUNT(*) INTO cnt FROM Class2 c2
    WHERE c2.attr2 > arg1;
  IF cnt > 0 THEN
    RETURN true;
  ELSE
    RETURN false;
  END IF;
```

```
END;
```

3.18 Specifying PARALLEL

The `PARALLEL` query hint specifies the degree of parallelism requested for the execution of a SQL statement. The actual number of threads used by a parallel query is determined at query plan execution initialization and is determined by the degree of parallelism and number of threads available in the SQL parallel processing pool of threads. The maximum degree of parallelism is set at the server level and defines the upper value which determines the maximum number of threads that are being used. You need to set the appropriate database configuration parameters to enable and to control the resources dedicated to parallel processing of queries.

```
SELECT PARALLEL(4)
  d.DepartmentName,
  d.employees.EmpId,
  d.employees.LastName,
  d.employees.GetAccrualsName(Bank::ListBankName(FALSE))
AS "Bank Name",
  d.employees.GetAccrualsQuantity(Bank::ListBankName(FALSE
)) AS Total
FROM
  Department d
ORDER BY
  d.DepartmentName,
  d.employees.LastName;
```

4 Using Numeric Values

4.1 Introduction

After reading this section, you should understand how Matisse analyzes arithmetic expressions and what data types result from different operations. You should also know how to:

- Use comparison operators
- Specify arithmetic operations on expressions
- Specify an interval test
- Negate expressions
- Use the `ANY` and `ALL` keywords with numeric values

4.2 Comparison Operators

- *Table 4.1* lists the various comparison operators that are available:

Table 4.1 Comparison Operators

Operator	Meaning
=	Equals
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

You can use these operators, for example, to compare an attribute value to a constant or to an expression as shown in the following section.

4.3 Bitwise Operators

◆ *Table 4.2* lists the various bitwise operators that are available:

Table 4.2 Bitwise Operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Invert bits
<<	Left shift
>>	Right shift

You can use these operators in the `SELECT` list and the `WHERE` clause as well as in block statement and SQL methods.

4.4 Performing Arithmetic Operations

Expressions and Arithmetic Operators

In Matisse SQL, an arithmetic expression can be any of the following:

```
expression
attribute
constant
value function
sum of expressions
product of expressions
quotient of expressions
difference between two expressions
```

An operation involving two expressions has the following syntax:

```
expression1 operator expression2
```

The binary operators that are valid are the multiplication operator `*`, the division operator `/`, the addition operator + and the subtraction operator `-`, which also acts as a negation operator when preceding a single expression.

The order of evaluation of an expression that contains two or more operators is determined by the hierarchy of operators. The sub-expressions within parentheses are evaluated first. The evaluation is performed in the following order, from left to right:

1. – negation operation
2. *, / multiplication and division

3. +, – addition and subtraction

Evaluating an Expression: An Example

To select the movies which would become longer than 90 minutes if their running time was increased by 15%, you could write a statement as shown below:

```
SELECT * FROM movie
 WHERE (runningTime * 115 / 100) >= 90
   AND runningTime <= 90;
```

Note that the expression to be evaluated in this request has the following format:

*(expression1 * constant1 / constant2)*

To evaluate this expression, Matisse SQL multiplies *expression1* by *constant1*. Then the product of this operation is divided by *constant2*.

A NULL value may result from processing an expression if one of the elements of the expression is not a numeric value type. Eventually, if the expression returns NULL, the first predicate in the above statement returns a logic value UNKNOWN since it cannot be evaluated.

4.5 Result Types from Arithmetic Expressions

The general format for an arithmetic operation between two expressions is the following:

expression1 operator expression2

In any arithmetic operation, the expressions to be operated on (the operands) must be numeric values.

The types resulting from the arithmetic operations are summarized in Table 4.3.

Table 4.3 Types Resulting from Arithmetic Operation

Operator	Expression1	Expression2	Result
+, –, *, /	LONG	LONG	LONG
	NUMERIC	NUMERIC	NUMERIC
	NUMERIC	LONG	
	LONG	NUMERIC	
	LONG	DOUBLE	DOUBLE
	DOUBLE	LONG	
	DOUBLE	NUMERIC	
	NUMERIC	DOUBLE	
	DOUBLE	DOUBLE	

When the operator is the division operator (/) and `expression2` has the value 0, the DIVISION_BY_ZERO error will be returned. For any operation, if the result is more than the precision that the type can hold, the NUMERICOVERFLOW error will be returned.

NOTE: When one of the terms of an arithmetic expression is `NULL`, the value of the resulting expression is `NULL`.

The negation operation produces the result types shown in *Table 4.4*.

Table 4.4 Type Resulting from the Negation Operation

Operator	Expression1	Result
–	FLOAT	FLOAT
	DOUBLE	DOUBLE
	SHORT	SHORT
	INTEGER	INTEGER
	LONG	LONG
	NUMERIC	NUMERIC

NOTE: The negation operation cannot be applied to BYTE type, since it does not allow negative number.

4.6 Performing an Interval Test

To test for values within an interval you can use a `BETWEEN .. AND` predicate as shown below:

```
expression
  [NOT] BETWEEN expression AND expression
```

Note that you can check that a value does *not* fall into an interval by inserting the keyword `NOT` immediately before `BETWEEN`.

To select the movies where the running time is between 90 minutes and 120 minutes, you can write the following statement:

```
SELECT * FROM movie WHERE runningTime
  BETWEEN 90 AND 120;
```

Note that the expression `BETWEEN 90 AND 120` is equivalent to the following:

```
WHERE runningTime >= 90
  AND runningTime <= 120;
```

4.7 Using the ANY and ALL Keywords

The `ANY` and `ALL` keywords let you compare a value to a set of values. The syntax is as follows:

```
expression comparison_operator
  {ANY | ALL} expressions
```

For more information please refer to the paragraph relating to the ANY and ALL keywords in *section 6.8, Using the ANY and ALL Keywords*.

5 Using Null Values

5.1 Introduction

This section explains how to use null values. After reading this section, you should know:

- What a null value is
- How to test for null values using the `IS NULL` keyword

5.2 What Is a Null Value?

In Matisse, the attribute of an object can be explicitly assigned a null value. An attribute for which no value has been assigned and for which there is no default value defined in the database schema is also seen as having a null value.

5.3 The IS NULL Keyword

You can check if an expression leads to a null value with the `IS NULL` keyword.

The syntax for evaluation of a null value is as follows:

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

The predicate:

```
expression IS NULL
```

is true if the result of the evaluation of the expression *`expression`* is null.

The predicate:

```
expression IS NOT NULL
```

is equivalent to the predicate:

```
NOT (expression IS NULL).
```

Example: Comparison with Null Values

The following request selects all the objects of class `movie` for which the attribute `runningTime` has a null value:

```
SELECT * FROM movie WHERE runningTime IS NULL;
```

A NULL value always leads to an UNKNOWN result when used directly in a comparison or any other operation. For example, when the runningTime value is null, the comparison in the following query will evaluate to UNKNOWN and thus will not return any object.

```
SELECT * FROM movie
 WHERE runningTime = NULL;
```

The behavior of IS [NOT] NULL is shown in *Table 5.1*.

Table 5.1 IS [NOT] NULL

Expression Value	IS NULL	IS NOT NULL
Null value	True	False
Valid value	False	True

6 Using Text Values

6.1 Introduction

Some topics covered in this section are similar to those presented in *section 4, Using Numeric Values*. The `ANY` and `ALL` keywords, for example, can also be used with numeric values.

After reading this section, you should know how to:

- Compare text values
- Specify wildcard characters in a pattern
- Specify an escape character with `ESCAPE` keyword
- Use the `ANY` and `ALL` keywords
- Select data by entry points

6.2 What Does Text Comparison Mean?

You can compare character strings with the same comparison operators that you use to compare numeric values. These operators are listed in *Table 6.1*.

Table 6.1 Text Comparison Operators

Operator	Meaning
=	Equals
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The comparison of a character string with a numeric value or any other non-character string value evaluates to `UNKNOWN`.

If the two character strings have the same characters at each position, they are equal. For example, the following character strings are equal:

```
'Rocky' = 'Rocky'
```

The following predicates evaluate to true:

```
'Rock' < 'Rocky'
'Mocky' < 'Rocky'
```

When two character strings are compared, the trailing blank spaces are not ignored but are taken into account. For example, the following predicates evaluate to false:

```
'Rocky ' = 'Rocky'
'Rocky ' = 'Rocky   '
```

The following predicate evaluates to true:

```
'Rocky ' = 'Rocky '
```

How Character Strings Are Compared

The character comparison between two strings is based on the ASCII (or EBCDIC) character values. A character string is greater than another character string when one or more of its characters has a higher ASCII (or EBCDIC) value than the character occupying the same position in the other character string.

All comparisons are performed on the basis of the number assigned to each character in the ASCII (or EBCDIC) character table shown in Table 6.2.

Table 6.2 ASCII Characters and Their Numeric Values

SP 32	0 48	@ 64	P 80	‘ 96	p 112
! 33	1 49	A 65	Q 81	a 97	q 113
" 34	2 50	B 66	R 82	b 98	r 114
# 35	3 51	C 67	S 83	c 99	s 115
$ 36	4 52	D 68	T 84	d 100	t 116
% 37	5 53	E 69	U 85	e 101	u 117
& 38	6 54	F 70	V 86	f 102	v 118
’ 39	7 55	G 71	W 87	g 103	w 119
(40	8 56	H 72	X 88	h 104	x 120
) 41	9 57	I 73	Y 89	i 105	y 121
* 42	: 58	J 74	Z 90	j 106	z 122
+ 43	; 59	K 75	[91	k 107	{ 123

Table 6.2 ASCII Characters and Their Numeric Values *(Continued)*

, 44	< 60	L 76	92	l 108	| 124
- 45	= 61	M 77	] 93	m 109	} 125
. 46	> 62	N 78	^ 94	n 110	~ 126
/ 47	? 63	O 79	_ 95	o 111	

The letter B, for example, has a number that is higher than A. In addition, lower case letters all have numbers that are greater than that of any upper case letter. While a has a number that is greater than those of all the other upper case letters, it has a number lower than that of b.

The following statements are equivalent and find all the movies other than Rocky with a running time greater than 90 minutes:

```
SELECT * FROM movie
  WHERE title <> 'Rocky'
        AND runningTime > 90;

SELECT * FROM movie
  WHERE NOT (title = 'Rocky'
            OR runningTime <= 90);
```

6.3 What Is a Pattern?

A *pattern* is a string of at most 255 characters, delimited by apostrophes (' '), that lets you specify the different characteristics you are searching for in a text string. These characteristics may include the following:

- Length of the string
- Constant characters in the string
- Variable (wildcard) characters in the string

6.4 How to Use the % Wildcard Character

A pattern accepts the same alphanumeric characters that can be used in any text string. In addition, a pattern may contain the following wildcard character:

```
%
```

The percent sign `%` is a wildcard character that represents any number of characters or no characters. Look, for example, at the following pattern:

```
'Ro%'
```

This pattern specifies the subset of character strings that starts with the characters *Ro*.

You can specify the `%` wildcard character at the beginning or in the middle of a character string, as shown in the examples below:

```
'%ocky'
'R%ky'
```

The second pattern specifies the subset of character strings beginning with the character `R` and ending with the characters k`y`.

6.5 How to Use the Underscore Wildcard Character

The underscore character _ functions similarly to the percent sign except that it represents *only one character*. The following example shows how this character is used:

```
'Rock_'
```

The above example specifies the subset of character strings containing five characters whose first 4characters make the substring 'Rock'.

6.6 Specifying a Pattern with the LIKE Keyword

When comparing two text strings, you can use the following syntax:

```
expression LIKE 'pattern'
```

The text string *expression* is compared to the master text string or *pattern*. The condition *expression* LIKE *'pattern'* is true if and only if the value of *expression* matches with *'pattern'*. Note that *pattern* needs to be a literal constant string.

An expression is comparable to a pattern only if it evaluates to a character string. If an expression evaluates to a value other than a character string, the comparison will evaluate to `UNKNOWN`. If the expression evaluates to a character string, the comparison will evaluate to `TRUE` or `FALSE`.

The following request selects all the movies whose names consist of at least two separate words, or consist of at least two separate words linked by a hyphen (-):

```
SELECT * FROM movie
 WHERE name LIKE '% %'
    OR name LIKE '%-%';
```

6.7 How to Use an Escape Character

The escape character is a character string composed of just one character. When it is defined, it becomes possible to use one of the wildcard characters as an ordinary character as long as you insert an escape character immediately before it.

For example, suppose you are looking for all the character strings that are 8 characters in length and begin with the characters '%ABC'. Since % already serves duty as the wildcard character, you cannot specify the ordinary character % with the wildcard character %. In this case, you must precede the character % with an escape character.

When you compare a text string with a pattern, you must define the escape character with the ESCAPE keyword, as shown in the following example:

```
expression LIKE '\%ABC%' ESCAPE '\'
```

This clause specifies all the character strings that are at least 4 characters in length and begin with the characters %ABC.

Some character strings that meet these criteria are listed below:

```
%ABC
%ABCDE
```

Note that if you want to specify the character used as the escape character in a search string, you must also precede it with itself, as shown below:

```
expression LIKE 'AB|%C||' ESCAPE '|'
```

This clause selects all the character strings that begin with the substring AB and end with the substring C|.

You cannot use the '\' for the search pattern escape character if some backslash escape sequences (\n, \t, \\, etc.) are used in the pattern.

```
expression LIKE '\\|%%' ESCAPE '|';
```

This clause specifies all the character strings begins with \%.

Some character strings that meet these criteria are listed below:

```
\%
\%\ABC
```

6.8 Using the ANY and ALL Keywords

The ANY and ALL keywords allows you to compare an expression to a set of expressions. They have the following syntax:

```
expression
  operator { ANY | ALL}
  ( expression [[,expression]... ] )
```

Quantified Comparison with the ANY Keyword

You can use the ANY keyword to formulate a quantified comparison between one character string and a set of character strings. Note that the operators used to check for equality or inequality between text values are the same as those discussed earlier in this section.

The comparison with the ANY keyword

- Is TRUE if the set of expressions contains at least 1 expression for which the comparison is true.
- Is FALSE if the comparison is false for every expression contained in the set of expressions.

Comparison with the ALL Keyword

The comparison with the ALL keyword

- Is TRUE if the comparison is true for every expression contained in the set of expressions.
- Is FALSE if there is at least one expression in the set of expressions for which the comparison is false.

Equivalent Comparisons

Certain negations of comparisons using ALL are equivalent to comparisons using ANY. *Table 6.3* lists these equivalences.

Table 6.3 Equivalent Expressions Using ANY and ALL

Expression Using NOT and ALL	Equivalent Expression
NOT (*expression* <> ALL *expressions*)	*expression* = ANY *expressions*
NOT (*expression* = ALL *expressions*)	*expression* <> ANY *expressions*
NOT (*expression* <= ALL *expressions*)	*expression* > ANY *expressions*
NOT (*expression* < ALL *expressions*)	*expression* >= ANY *expressions*
NOT (*expression* >= ALL *expressions*)	*expression* < ANY *expressions*
NOT (*expression* > ALL *expressions*)	*expression* <= ANY *expressions*

Alternate Syntax

The keyword `IN` can be used instead of `= ANY` and the keywords `NOT IN` can be used instead of `<> ALL`. Note, for example, the statement:

```
SELECT * FROM movie
 WHERE title = ANY('Rocky', 'Grease');
```

is equivalent to the statement:

```
SELECT * FROM movie
 WHERE title IN LIST(STRING) ('Rocky', 'Grease');
```

In the same way, the statement:

```
SELECT * FROM movie
 WHERE title <> ALL ('Rocky','Grease');
```

is equivalent to the request:

```
SELECT * FROM movie
 WHERE title NOT IN LIST(STRING) ('Rocky','Grease');
```

Examples

The following statement selects the objects of the class `movie` whose value for the attribute `title` is *Rocky*, *Grease*, or *Casper*:

```
SELECT * FROM movie
  WHERE title =
    ANY('Rocky','Grease','Casper')
```

This request is equivalent to the following statement:

```
SELECT * FROM movie
  WHERE NOT
    (title <> ALL('Rocky','Grease','Casper'))
```

Both statements are equivalent to the following one:

```
SELECT * FROM movie
  WHERE title = 'Rocky' OR name = 'Grease'
    OR title = 'Casper'
```

6.9 Selecting Objects by Entry Points

The Entry Point Dictionaries of Matisse offer an efficient mechanism to implement a full text search capability.

When an Entry Point Dictionary is defined in the database schema for a given attribute, the creation of an object and the subsequent updates of the attribute automatically populate the dictionary with a list of keywords generated from the new value of the attribute. These keywords are called entry points.

Entry Point Dictionaries can be accessed to retrieve objects either from Matisse SQL or language bindings, e.g., the Java binding.

Exact Match Search

To search for objects through an entry point with an exact match, you must use the following syntax:

```
[NOT] [<navigation>.]ENTRY_POINT (entry_point_dictionary)
  {=| <>} 'entry_point'
```

Assuming that you have defined on the attribute `synopsis` for the class `movie` an Entry Point Dictionary that indexes every word in a text string, you may select the movies whose synopsis contains the word “adventure” with the following statement:

```
SELECT * FROM movie
 WHERE ENTRY_POINT(MovieSynopsisDict) = 'adventure';
```

You can combine entry points predicates with any other predicates by using `OR` and `AND` keywords, for instance:

```
SELECT * FROM movie
 WHERE ENTRY_POINT(MovieSynopsisDict) = 'adventure'
       OR ENTRY_POINT(MovieSynopsisDict) = 'lost'
       AND title <> 'Rocky';
```

ENTRY_POINT() may be preceded by a relationship navigation, for example:

```
SELECT * FROM movie m
 WHERE m.starring.ENTRY_POINT(LastNameDict) = 'Cruise';
```

Pattern Matching

To search for objects through an entry point with pattern matching, you must use the following syntax:

```
ENTRY_POINT(entry_point_dictionary) [NOT] LIKE
     [ ESCAPE 'escape-char' ]
```

The same rules as the ones described for the clause `LIKE` apply for the wildcard and escape characters.

You may select the movies whose synopsis contains the pattern ‘adventure%’ for instance to qualify objects containing either ‘adventure’ or ‘adventurers’:

```
SELECT * FROM movie
 WHERE ENTRY_POINT(MovieSynopsisDict) LIKE 'adventure%';
```

7 Using Relationships

7.1 Introduction

This section describes how to navigate through relationships within an SQL statement. After reading this section, you should know:

- What a relationship is
- How to use the `IN` keyword
- How to use relationships in the Select-list and the where clause

7.2 What Is a Relationship?

In Matisse a relationship defines a link between an object and other objects. From a given object, called a predecessor, the objects that a relationship points to are referred to as the successors of the object through that relationship. The successors of a relationship can be either a set of objects or a `NULL` value when there is no successor for the relationship.

The successor objects are either ordered or unordered. When a relationship is defined as LIST, its successors are ordered. When a relationship is defined as SET, its successors are unordered.

7.3 Positional Access

A successor object at a specific position in a relationship can be accessed using the positional access syntax:

```
relationship_name(number)
```

For example, the following query statement returns movies whose first starring's last name is Brody:

```
SELECT m.Name FROM movie m
 WHERE m.Starring(1).LastName = 'Brody';
```

The first successor object is at position 1. When the number is out of range, i.e., more than the number of successors or less than 1, the positional access expression returns NULL.

For an unordered relationship, i.e., defined as SET relationship, positions of successors are system-defined, and not guaranteed to be the same every time.

7.4 Navigational Queries

You can navigate through the relationships within the `Select-list` or the WHERE clause.

Using a Single Relationship in the Select-list

The syntax of a relationship expression is as follows:

```
[{class | alias }.]navigation.{attribute|*}
```

With navigation such as:

```
navigation ::=
  relationship[.({CLASS | ONLY} successor_class)]
  [.relationship[.({CLASS | ONLY} successor_class)] ...]
```

For instance to retrieve the directors of the movies with a title like “Rocky%”, you would write the following statement:

```
SELECT directedBy.* FROM movie
 WHERE title LIKE 'Rocky%';
```

The same statement with full class qualification would be expressed as follows:

```
SELECT movie.directedBy.lastname
  FROM movie
 WHERE movie.title LIKE 'Rocky%';
```

You can also filter the results from a class or subclass of the successors by specifying a successor class in the navigational expression using the keyword CLASS. For instance if you want to find the movie directors who are also starring in some movies, you could write the following query:

```
SELECT m.starring.(CLASS movieDirector).lastname
  FROM movie m;
```

If you filter the successors using the keyword ONLY instead of CLASS, the result includes only the ‘proper’ instances of the class, i.e., excluding the instances of its subclasses. For example, the next query returns the starring actors of each movie who are NOT movie directors:

```
SELECT m.starring.(ONLY artist).lastname
  FROM movie m;
```

Using Relationships and Other Columns in the Select-list

The relationships with multiple successors are “exploded” in the projection result in a similar way a relational join would do. For instance, a movie starring two actors would display a result as follows:

```
SELECT m.title,
       m.starring.lastname AS Starring
  FROM movie m
 WHERE m.title = 'Titanic';
```

Result:

```
Title         Starring
-------       ---------
Titanic       DiCaprio
Titanic       Winslet
```

Using a Relationship in the WHERE Clause

You can access attributes through relationship navigation in the predicate expressions in the WHERE clause with the following syntax:

```
[{class | alias }.]navigation.attribute
```

When a relationship is multi-valued, which means that several objects can be reached through this relationship, the comparison with an other expression is true if **any** of the objects evaluates to true.

For instance, to retrieve the movies where any actor has a last name starting with ‘S’, you would write the following statement:

```
SELECT * FROM movie
 WHERE starring.lastname LIKE 'S%';
```

You can combine this with a relationship in the Select-list. For instance, the following query is valid and returns the directors for the movies that qualify:

```
SELECT directedBy.* FROM movie
 WHERE starring.lastname LIKE 'S%';
```

The same query expressed with full class qualification is expressed as follows:

```
SELECT movie.directedBy.* FROM movie
 WHERE movie.starring.lastname LIKE 'S%';
```

Relationship COUNT

You can also check for the cardinality of a relationship with the built-in function `COUNT` which can be expressed with the following syntax:

```
COUNT (relationship [({CLASS | ONLY}.successor_class)])
```

For instance, to check for the movies starring more than 10 actors, you could write the following statement:

```
SELECT * FROM movie
  WHERE COUNT(starring) > 10;
```

To check for the movies where one movie director is starring:

```
SELECT * FROM movie
 WHERE COUNT(starring.(CLASS movieDirector)) = 1;
```

To check for the movies starring two actors excluding movie directors:

```
SELECT * FROM movie
 WHERE COUNT(starring.(ONLY artist)) = 2;
```

Dealing with Empty Relationships

When a relationship has no successor in Matisse, it is always implemented as a `NULL` relationship. Consequently, the following query has a correct syntax, but it will never retrieve any object:

```
SELECT * FROM movie
 WHERE COUNT(starring) = 0;
```

This query should be rewritten as follows in order to find the movies for which there is no actor:

```
SELECT * FROM movie
 WHERE starring IS NULL;
```

If you want to retrieve the movies where there are between 0 and 5 actors, you could express it as follows:

```
SELECT * FROM movie
 WHERE COUNT(starring) <= 5
    OR starring IS NULL;
```

7.5 The IN Keyword

By using the `IN` keyword, you can select objects based on the evaluation of the inclusion of two sets of objects. The sets of objects can be either a selection result obtained with an other statement or the objects that are successors through a relationship.

The keyword `IN` has the following syntax:

```
{ALL | ANY} set1 IN set2
```

For each object belonging to *`set1`*, the keyword `IN` checks whether or not it also belongs to *`set2`*.

If the keyword *`ALL`* is specified, the inclusion is true if all the objects of *`set1`* belong to *`set2`*. If *`ANY`* is specified, the inclusion is true if any of the objects of *`set1`* belong to *`set2`*.

For instance, to select the movies where all the directors are also starring in the movie you would use the following command:

```
SELECT * FROM movie
 WHERE ALL directedBy IN starring;
```

To select the movies where any director is also starring in the movie you would use one of the following command:

```
SELECT * FROM movie
 WHERE ANY directedBy IN starring;
```

You can combine with the keyword NOT. For instance, to select the movies where no director is starring in the movie:

```
SELECT * FROM movie
 WHERE NOT ANY directedBy IN starring;
```

Comparing with a List of Successors

In addition to comparing the successors of an object through different relationships, you can also compare successors to the result of a previous statement execution.

For example, if you want to know which movies have a director whose name starts with 'R' and is also starring in the movie, you can first select the directors by their name:

```
SELECT REF(m) FROM movieDirector m
 WHERE m.lastname LIKE 'R%'
  INTO mDirectors;
```

Then, you get the movies with the following request:

```
SELECT * FROM movie
 WHERE ANY starring IN mDirectors;
```

Note that with the navigation capability of Matisse SQL, the two queries could be written in only one statement, without the need to use an intermediate result:

```
SELECT * FROM movie
 WHERE directedBy.lastname LIKE 'R%'
   AND ANY directedBy IN starring;
```

8 Version Travel

8.1 Introduction

With Matisse you can save and query consistent versions of the database; a saved version can be accessed until it is explicitly deleted. This section describes how to select objects which have been updated, inserted, or deleted across two different database versions.

For additional details on accessing database versions, see *section 9, Managing Transactions and Versions*.

8.2 Specifying a Version Travel Query

You can specify the type of version travel operation in the `FROM` list, with the following syntax:

```
FROM UPDATED
    ( { [ONLY]class | selection }, {BEFORE | AFTER} version )
FROM INSERTED
    ( { [ONLY]class | selection }, AFTER version )
FROM DELETED
    ( { [ONLY]class | selection }, BEFORE version )
```

When using the keyword `BEFORE`, you may specify *`version`* with either a version name or `CURRENT` for the most recent version.

For instance, if you save a version every day for seven days, you may want to find the objects updated between the versions named `day1` and `day2`, with `day1` older than `day2`. For this, you first set the access mode as of `day2`, then you can execute a version travel query as follows:

```
SET TRANSACTION READ ONLY day2;
SELECT * FROM UPDATED (movie, AFTER day1);
```

The objects that you have selected will be read in the version context for the newer version `day2`.

You can also use the keyword `BEFORE` to retrieve the objects as of the older version `day1`. Note that in this case the `WHERE` clause is evaluated in the `day1` context.

```
SET TRANSACTION READ ONLY day1;
```

```
SELECT * FROM UPDATED (movie, BEFORE day2) WHERE rating LIKE
'PG%';
```

For inserted objects, you can only access the objects as of the newer version.

```
SET TRANSACTION READ ONLY day2;
SELECT * FROM INSERTED (movie, AFTER day1);
```

For deleted objects, you can only access the objects as of the older version.

```
SET TRANSACTION READ ONLY day1;
SELECT * FROM DELETED (movie, BEFORE day2);
SELECT * FROM DELETED (movie, BEFORE CURRENT);
```

9 Managing Transactions and Versions

9.1 Introduction

This section describes how to access or modify data in a Matisse database. After reading this section, you should be able to perform the following operations:

- Obtain read-only access on a database
- Obtain read/write access on a database
- Commit a transaction
- Cancel a transaction

9.2 Starting a Version Access

To obtain read-only access on the current connection, you must use the following syntax:

```
SET TRANSACTION READ ONLY [savetime]
```

To obtain read only access at the latest logical time, you can use the following command:

```
SET TRANSACTION READ ONLY
```

To obtain read only access for a particular savetime, which is a consistent "snapshot" of the database at a particular time (for more information about savetime, refer to the "*Getting Started with Matisse*" document), you must specify the savetime, as in the following example:

```
SET TRANSACTION READ ONLY August2006;
```

The savetime specified can be either the fully qualified name generated by Matisse upon commit, or only the prefix as shown on the above example.

Using a fully qualified name allows you to identify a savetime without ambiguity when several savetimes have been generated with the same prefix. For instance, if we suppose that the version `August2006` was saved at the logical time `2A` (in hexadecimal), the statement from the previous example could be expressed as follows:

```
SET TRANSACTION READ ONLY August20060000002A
```

NOTE: If you perform a `SELECT` on a connection where you have not previously set an access mode, the request will be executed in read-only version mode on the latest version of the database.

9.3 Ending a Version Access

To end a read only access to database, you can use the following syntax:

```
ROLLBACK [WORK]
```

The optional keyword WORK has no effect on the execution. In either case, the current version access is terminated.

9.4 Starting a Transaction

You may want to start a transaction on the current connection explicitly. This may be necessary if you have previously set the connection to version access.

To start a transaction, you can use the following syntax:

```
SET TRANSACTION READ WRITE [priority]
```

The optional argument *`priority`* lets you specify the priority of the transaction. Permitted values for this argument are integers in the range 0 (lowest priority) to 9 (highest priority). By default, the priority is 0.

For example, to start a transaction with the highest priority, you would write the following command:

```
SET TRANSACTION READ WRITE 9;
```

9.5 Committing a Transaction

To validate a transaction, you can use the following syntax:

```
COMMIT [WORK] [VERSION savetime_prefix]
```

The following commands are equivalent:

```
COMMIT
COMMIT WORK
```

The optional argument *`savetime_prefix`* enables you to save the logical time resulting from the transaction as a savetime. (The actual identifier of the savetime will be made up of the prefix followed by the logical time that corresponds to the transaction.)

To commit a transaction and save the corresponding logical time as a savetime, you can use a command like the following:

```
COMMIT WORK VERSION August2006;
```

This command commits the transaction. The logical time resulting from the transaction will be saved in a savetime. The prefix of this savetime will be `August2006`.

NOTE: The name of the full savetime is output by `COMMIT` command when it has concluded successfully.

Note that a savetime prefix cannot exceed 20 characters in length.

9.6 Cancelling a Transaction

There are times when you may want to cancel the modifications of a transaction. To do this, use the `ROLLBACK` command. This command has the following syntax:

```
ROLLBACK [WORK]
```

You can use the command `ROLLBACK` by itself or followed by the keyword `WORK`. In either case the current transaction is cancelled. The following commands are equivalent:

```
ROLLBACK
ROLLBACK WORK
```

10 SQL Functions

This section explains how to use Matisse SQL functions. Matisse SQL has many built-in functions that are applicable to various data types. You can use these functions anywhere expressions are allowed.

After reading this section, you should know how to use:

- Expressions functions
- Character string functions
- List functions
- Set functions (aggregate functions)
- Set functions for relationship aggregation
- Datetime functions
- Conversion functions
- Numeric functions

10.1 Expressions Functions

The following functions simplify the management of `NULL` values.

- `COALESCE`
- `NULLIF`

COALESCE

Syntax

```
COALESCE(expr1, expr2 [, expr3 ... , expr15])
```

Purpose

Evaluates the arguments in order and returns the current value of the first expression that initially does not evaluate to `NULL`. If all arguments are `NULL`, `COALESCE` returns `NULL`.

Arguments

expr1, expr2 [, *expr3* ... , *expr15*]

These can be Matisse attributes or any expressions. The function requires at least 2 arguments.

Example

```
sql> SELECT a.firstName, a.lastName,
  2>       COALESCE(a.firstName, a.lastName) AS firstNotNull
  3>    FROM artist AS a;
```

NULLIF

Syntax `NULLIF(expr1, expr2)`

Purpose Returns the first expression if the two expressions are not equal. If the expressions are equal, `NULLIF` returns a null value of the type of the first expression.

Arguments *`expr1, expr2`*

These can be Matisse attributes or any expressions.

Example
```
sql> SELECT AVG(NULLIF(runningTime, 0)) FROM movie;
```

10.2 Character String Functions

The following character string functions return character string. The type of returned character string is `STRING`.

- `CONCAT`
- `LEFT`
- `LOWER`
- `LPAD`
- `LTRIM`
- `REPLACE`
- `REPLICATE`
- `REVERSE`
- `RIGHT`
- `RPAD`
- `RTRIM`
- `SUBSTR (SUBSTRING)`
- `TRIM`
- `UPPER`

The following character string functions return numeric values. The return type is `INTEGER`.

- `INSTR`
- `LENGTH (CHAR_LENGTH)`
- `LOCATE`

CONCAT

Syntax `CONCAT(string1, string2)`

Purpose Concatenates two argument strings and returns the result.

Arguments *string1*, *string2*

These can be Matisse attributes or any expressions that return a character string. If one of the arguments is NULL or NULL pointer and the other argument is a valid string, the valid string is returned.

Example

```
sql> SELECT a.firstName, a.lastName,
  2>       CONCAT(a.firstName, a.lastName) concatenated
  3>   FROM artist AS a;
firstName     lastName      concatenated
------------- ------------- ----------------
Leonardo      DiCaprio      LeonardoDiCaprio
```

INSTR

Syntax `INSTR(string1, string2 [, n [, m]])`

Purpose Returns the character position in *string1* where *string2* appears.

Arguments

string1

The character string that you want to search. If this is not a valid character string, NULL is returned.

string2

The character string that you want to find in *string1*. If this is not a valid character string, NULL is returned.

`n`

The character position where the function starts to search. If, for example, `n` is 2, the search begins from the second character in *string1*. If `n` is negative, counts backward from the end of *string1* and searches backward from that position. If `n` is 0, it is treated as 1. The default value is 1.

`m`

When *string2* appears in *string1* more than once, `m` specifies which occurrence you want to find. If `m` is not positive, NULL is returned. The default value is 1.

Description The return value is relative to the beginning of `string1` regardless of the value of `n`. When *string2* is not found in *string1* under the specified condition, the function returns 0. When *string2* is an empty string and *string1* is a valid character string, the result is non-zero number.

Example

```
sql> SELECT INSTR('MATISSE MATINEE', 'MAT', 1, 2) FROM ...;
------------
```

```
            9

sql> SELECT INSTR('MATISSE MATINEE', 'MAT', -1) FROM ...;
------------
            9
```

LEFT

Syntax

```
LEFT(string,len)
```

Purpose

Returns the leftmost *len* characters from *string*, or NULL if any argument is NULL.

Arguments

string

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

`len`

The maximum number of characters returned. If the argument is NULL, this function returns NULL.

Example

```
sql> SELECT m.title, LEFT(m.title,2) AS len FROM movie AS m;
title          left
-------------- --------
Rocky                 Ro
```

LENGTH

Syntax

```
LENGTH(string)
CHAR_LENGTH(string)
```

Purpose

Returns the number of characters in a string.

Arguments

string

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

Description

If the argument is an empty string, the function returns 0. If the argument is a NULL pointer, the function returns NULL.

Example

```
sql> SELECT m.title, LENGTH(m.title) t_length FROM movie m;
title         t_length
----------    ---------
Rocky                 5
```

LOCATE

Syntax

```
LOCATE(substr, string [,n])
```

Purpose Returns the position of the first occurrence of *substr* in *string*, starting at position *n*. Returns 0 if *substr* is not in *string*.

Arguments

`substr`

The character string that you want to find in *string*. If this is not a valid character string, NULL is returned.

`string`

The character string that you want to search. If this is not a valid character string, NULL is returned.

`n`

The character position where the function starts to search. If, for example, `n` is 2, the search begins from the second character in *string*. If `n` is negative, counts backward from the end of *string* and searches backward from that position. If `n` is 0, it is treated as 1. The default value is 1.

Example

```
sql> SELECT LOCATE('MAT', 'MATISSE MATINEE') FROM ...;
locate
------------
           1
```

LOWER

Syntax

```
LOWER(string)
```

Purpose Returns a string in which all characters are converted to lowercase.

Arguments

`string`

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

Example

```
sql> SELECT m.title, LOWER(m.title) low FROM movie m;
title     low
--------  ------
Rocky     rocky
```

LPAD

Syntax

```
LPAD(string,len,padstr)
```

Purpose Returns the string *string*, left-padded with the string *padstr* to a length of *len* characters. If *string* is longer than *len*, the return value is shortened to *len* characters.

Arguments *string*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

`len`

The number of characters returned. If the argument is NULL, this function returns NULL.

padstr

A set of characters to pad *string* with.

Example

```
sql> SELECT m.title, LPAD(m.title,8,'.') FROM movie AS m;
title         lpad
------------- --------------
Rocky         ...Rocky
```

LTRIM

Syntax

```
LTRIM(string1 [, string2])
```

Purpose Removes characters from the left of *string1*, with all the lifetimes characters that appear in *string2* removed, and returns the result.

Arguments *string1*

The string characters from which you want to remove leading characters. This can be a Matisse attribute or any expression that returns a character string.

string2

A set of characters to be removed from *string1*. When this is omitted, it is substituted by a single space.

Description Trimming terminates when a character that does not appear in *string2* is encountered. If all characters in *string1* are removed, an empty string is returned. If `string1` is an empty string, an empty string is returned. If *string1* is a NULL pointer, NULL is returned.

Example

```
sql> SELECT LTRIM('baacde', 'ab') trimmed FROM ...;
trimmed
-----------
cde
```

REPLACE

Syntax `REPLACE(string,fromstr,tostr)`

Purpose Returns *`string`* with all occurrences of the *`fromstr`* replaced by *`tostr`*.

Arguments *`string`*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

`fromstr`

The character string that you want to replace.

`tostr`

The character string that you are replacing with.

Example

```
sql> SELECT REPLACE('matisse', 'm', 'M') FROM ...;
replace
-------------
Matisse
```

REPLICATE

Syntax `REPLICATE(string,n)`

Purpose Returns a string consisting of the string *`string`* repeated *`n`* times. If *`n`* is less than 1, returns an empty string. Returns NULL if *`string`* or *`n`* are NULL.

Arguments *`string`*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

`n`

The number of times the string is repeated. If the argument is NULL, this function returns NULL.

Example

```
sql> SELECT REPLICATE('matisse', 2) FROM ...;
replicate
--------------
matissematisse
```

REVERSE

Syntax `REVERSE(string)`

Purpose Returns *string* with the order of the characters reversed.

Arguments *string*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

Example

```
sql> SELECT REVERSE('matisse') FROM ...;
reverse
--------------
essitam
```

RIGHT

Syntax `RIGHT(string,len)`

Purpose Returns the rightmost *len* characters from *string*, or NULL if any argument is NULL.

Arguments *string*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

`len`

The maximum number of characters returned. If the argument is NULL, this function returns NULL.

Example

```
sql> SELECT m.title, RIGHT(m.title,2) AS len FROM movie AS
m;
title         right
------------- --------
Rocky                 ky
```

RPAD

Syntax `RPAD(string,len,padstr)`

Purpose Returns the string *string*, right-padded with the string *padstr* to a length of *len* characters. If *string* is longer than *len*, the return value is shortened to *len* characters.

Arguments *string*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

`len`

The number of characters returned. If the argument is NULL, this function returns NULL.

padstr

A set of characters to pad *string* with.

Example

```
sql> SELECT m.title, RPAD(m.title,8,'.') AS low FROM movie
AS m;
title         rpad
------------- --------------
Rocky         Rocky...
```

RTRIM

Syntax

```
RTRIM(string1 [, string2])
```

Purpose

Removes characters from the right of *string1*, with all the rightness characters that appear in *string2* removed, and returns the result.

Arguments

string1

The string characters from which you want to remove some trailing characters. This can be a Matisse attribute or any expression that returns a character string.

string2

A set of characters to be removed from `string1`. When this is omitted, it is substituted by a single space.

Description

Trimming terminates when a character that does not appear in *string2* is encountered. If all characters in *string1* are removed, an empty string is returned. If `string1` is an empty string, an empty string is returned. If *string1* is a NULL pointer, NULL is returned.

Example

```
sql> SELECT RTRIM('abc d ef', 'def ') trimmed FROM ...;
trimmed
------------
abc
```

SUBSTR

Syntax

```
SUBSTR(string, m [, n])
SUBSTRING(string, m [, n])
```

Purpose

Returns a portion of character string, beginning at position *m*, *n* characters long.

Arguments

string

The input character string. This can be a Matisse attribute or any expression that returns a character string. If this is not a valid character string, NULL is returned.

m

The position in string where the extraction begins. If *m* is positive, the function counts from the beginning of string. If *m* is greater than the length of string, an empty string is returned. If *m* is 0, it is treated as 1. If *m* is negative, the function counts backwards from the end of string. If the length of string plus *m* is less than or equal to 0, the position is treated as the beginning of string.

n

The number of characters to be extracted. If *n* is omitted, returns all characters beginning from the position specified by *m* to the end of string. If *n* is less than 1, an empty string is returned. If string does not have *n* characters after position *m*, returns all characters from the position *m* to the end of string.

Example

```
sql> SELECT SUBSTR('MATISSE SQL', 6) extracted FROM ...;
extracted
--------------
SE SQL

sql> SELECT SUBSTR('MATISSE SQL', -6, 2) extracted FROM ...;
extracted
--------------
SE
```

TRIM

Syntax

```
TRIM(string1 [, string2])
```

Purpose

Returns `string1` with `string1` leading an trailing characters that appear in `string2` removed. The default value for `string2` is a single space (' ').

Arguments

string1

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

string2

A set of characters to be removed from `string1`. When this is omitted, it is substituted by a single space.

Example

```
sql> SELECT TRIM('babacdeabba', 'ab') AS trimmed FROM ...;
trimmed
-------------
cde
```

UPPER

Syntax `UPPER(string)`

Purpose Returns a string in which all characters are converted to uppercase.

Arguments *`string`*

This can be a Matisse attribute or any expression that returns a character string. If the argument is not a character string, this function returns NULL.

Example

```
sql> SELECT m.title, UPPER(m.title) up FROM movie m;
title         up
------------ -----------
Rocky         ROCKY
```

10.3 List Functions

Matisse provides SQL `list` functions that allow you to access elements in a list, to get the number of elements, or to do aggregate calculations on a list. The list types that can be used with the SQL `list` functions are:

- `LIST(SHORT)`
- `LIST(INTEGER)`
- `LIST(LONG)`
- `LIST(FLOAT)`
- `LIST(DOUBLE)`
- `LIST(BOOLEAN)`
- `LIST(DATE)`
- `LIST(TIMESTAMP)`
- `LIST(INTERVAL)`
- `LIST(STRING)`
- `LIST(NUMERIC(p, s))`

The followings are the functions that works with list types:

- AVG
- MIN
- MAX
- SUM
- COUNT
- ELEMENT

- SUBLIST

AVG

Syntax
```
AVG(list)
```

Purpose Returns the average value of all the elements in a list.

Argument *list*

A list. If this argument is not a list, NULL is returned. The function accepts the numeric list types as well as `LIST(INTERVAL)`.

Description The return type of the function is `DOUBLE` regardless of the type of list except for `LIST(INTERVAL)`, in which case `INTERVAL` is returned, and `LIST(NUMERIC)`, in which case `NUMERIC` is returned.

Example
```
sql> SELECT AVG(LIST(10, 20, 40)) average FROM ...;
average
-----------
    23.3333
```

ELEMENT

Syntax
```
ELEMENT(list, n)
```

Purpose Returns an element at position `n` in `list`.

Argument *list*

A list. If this argument is not a list, `NULL` is returned. The function accepts all types of `list`.

n

The position at which you want to get an element. If *n* is 0, it is treated as 1. If *n* is negative, the function counts backwards from the end of the list. If *n* is out of bounds of `list`, NULL is returned.

Example
```
sql> SELECT ELEMENT(LIST(INTEGER)(10, 20, 30, 40), 2)
  2>   FROM ... ;
-------------
            20

sql> SELECT ELEMENT(LIST(INTEGER)(10, 20, 30, 40), -2)
  2>   FROM ... ;
-------------
```

```
           30
```

MAX

Syntax `MAX(list)`

Purpose Returns the maximum value of the elements in a list.

Argument *`list`*

A list. If this argument is not a list, NULL is returned. The function accepts the numeric list types as well as `LIST(DATE)`, `LIST(TIMESTAMP)`,and `LIST(INTERVAL)`.

Example

```
sql> SELECT MAX(LIST(INTEGER)(10, 20, 30, 40))
  2>   FROM ... ;
-------------
           40
```

MIN

Syntax `MIN(list)`

Purpose Returns the minimum value of the elements in a list.

Argument *`list`*

A list. If this argument is not a list, NULL is returned. The function accepts the same types as `MAX`.

Example

```
sql> SELECT MIN(LIST(INTEGER)(10, 20, 30, 40))
  2>   FROM ... ;
-------------
           10
```

SUBLIST

Syntax `SUBLIST(list, n [, m])`

Purpose Returns a portion of the list, beginning at position `n`, `m` elements long.

Argument *`list`*

A list. If this argument is not a list, NULL is returned.

`n`

The position in *list* where the extraction begins. If *n* is positive, the function counts from the beginning of *list*. If *n* is greater than the number of elements in *list*, NULL is returned. If *n* is 0, it is treated as 1. If *n* is negative, the function counts backwards from the end of *list*. If the number of elements in *list* plus *n* is less than or equal to 0, the position is treated as the beginning of *list*.

m

The number of elements to be extracted. If *m* is omitted, returns all elements beginning from the position specified by *n* to the end of *list*. If *m* is less than 1, an empty list is returned. If *list* does not have *m* elements after the position *n*, returns all elements from the position *n* to the end of *list*.

Example

```
sql> SELECT SUBLIST(LIST(INTEGER)(10, 20, 30, 40), 2)
  2> as ranking, title
  3>   FROM movie;
ranking   title
--------- ---------
       20 Rocky
       30 Rocky
       40 Rocky
```

Example

```
sql> SELECT SUBLIST(LIST(INTEGER)(10, 20, 30, 40), -3, 2)
  2> as ranking, title
  2>   FROM movie;
ranking   title
--------- ---------
       20 Rocky
       30 Rocky
```

SUM

Syntax

```
SUM(list)
```

Purpose

Returns the sum of the values in *list*.

Argument

list

A list. If this argument is not a list, NULL is returned. The function accepts the numeric list types as well as `LIST(INTERVAL)`.

Example

```
sql> SELECT SUM(LIST(INTEGER)(10, 20, 30, 40)) total
  2>   FROM ... ;
total
-------------
          100
```

COUNT

Syntax `COUNT(list)`

Purpose Returns the number of elements in *list*.

Argument *list*

A list. If this argument is not a list, NULL is returned.

Description If *list* is an empty list, the function returns 0. Note that if an attribute has not been assigned a value yet, that is, the attribute's value is `NULL`, this function does not return 0, but it returns NULL.

LIST

Syntax `LIST(type)({constant1 [, constant2, ...]})`

Purpose Constructs a new constant list and returns it. See *section 10.3, List Functions*, for more information.

Example

```
INSERT INTO boxOffice (topReceipts)
  VALUES (LIST(NUMERIC(10, 2))(34.5, 20.0, 8.9, 3.3, 2.1));
```

10.4 Set Functions

Matisse provides the following set functions to summarize data from multiple objects as a result of SQL query execution. These functions work only in SQL projection. You cannot put more than one set function in an SQL statement in this release.

- AVG
- COUNT
- MAX
- MIN
- SUM

AVG

Syntax `AVG ([class.|alias.]attribute)`

Purpose Returns the average value for an attribute from the set of objects which qualify the query.

Argument *attribute*

Numeric types and `INTERVAL` are accepted. Note that if this argument is a type of list, the function acts as a `list` function.

Description The result types are as follows:

```
Argument                        Result
---------------                 --------------
Any numeric type except NUMERIC DOUBLE
NUMERIC                         NUMERIC
INTERVAL                        INTERVAL
```

Example To get the average running time:

```
SELECT AVG(runningTime) FROM movie;
```

COUNT

Syntax `COUNT ([{class | alias}.]*)`

Purpose Returns the number of objects which qualify the query.

Example For instance, the following queries are equivalent to get the count of all the movies in the database:

```
SELECT COUNT(*) FROM movie;
SELECT COUNT(movie.*) FROM movie;
SELECT COUNT(m.*) FROM movie m;
```

To get the count of all the movie directors who are directing certain movies:

```
SELECT SUM(COUNT(m.movieDirector)) FROM movie m;
```

MAX

Syntax `MAX (attribute)`

Purpose Returns the maximum value for an attribute from the set of objects which qualify the query.

Argument *attribute*

Numeric types, `DATE`, `TIMESTAMP`, and `INTERVAL` are accepted. Note that if this argument is a list, the function acts as a `list` function.

Description The result types are as follows:

```
Argument             Result
---------------      --------------
Any numeric type     Same type
DATE                 DATE
TIMESTAMP            TIMESTAMP
INTERVAL             INTERVAL
```

MIN

Syntax

```
MIN (attribute)
```

Purpose Returns the minimum value for an attribute from the set of objects which qualify the query.

Argument *attribute*

Numeric types, DATE, TIMESTAMP, and INTERVAL are accepted. Note that if this argument is a list, the function acts as a list function.

Description The result types are as follows:

```
Argument             Result
---------------      --------------
Any numeric type     Same type
DATE                 DATE
TIMESTAMP            TIMESTAMP
INTERVAL             INTERVAL
```

SUM

Syntax

```
SUM (attribute)
```

Purpose Returns the sum value for an attribute from the set of objects which qualify the query.

Argument *attribute*

Numeric types and INTERVAL are accepted. Note that if this argument is a list, the function acts as a list function.

Description

```
The result types are as follows:
Argument             Result
---------------      --------------
Any numeric type     Same type
INTERVAL             INTERVAL
```

Example To get the sum of running time:

```
SELECT SUM(runningTime) FROM movie;
```

10.5 Set functions for relationship aggregation

Matisse provides the following set functions to summarize data from multiple successor objects of a relationship.

- AVG
- COUNT
- MAX
- MIN
- SUM

Suppose we have two classes `Department` and `Employee` where `Department` has a relationship `employees` referencing a set of `Employee` objects, and `Employee` has an attribute `salary` of type NUMERIC.

The next SELECT statement returns the name of each department and the total salary of employees working for the department:

```
SELECT d.name, SUM (d.employees.salary) FROM Department d;
name            sum
--------------- -------------
Engineering        3467600.00
Marketing           944890.00
```

The SUM function here sums salaries of all the employees of a department, i.e., the function aggregates some data of all the successor objects of a relationship.

The general form is:

```
SetFunction (<navigation>.attribute)
<navigation> ::=
  relationship[.({CLASS | ONLY} successor_class)]
  [.relationship[.({CLASS | ONLY} successor_class)] ...]
```

In order to use the functions for relationship aggregation, `attribute` needs to be of atomic type, not of list type. If `attribute` is of list type, e.g., LIST(INTEGER), the function works as list function explained in the *section 10.3, List Functions*, and no aggregation on relationship happens.

If no successor object is found for a relationship, these set functions return NULL.

AVG

Syntax `AVG ([class.|alias.]navigation.attribute)`

Purpose Returns the average value for `attribute` from the set of successor objects accessible through `navigation`.

Argument *`attribute`*

Numeric types and `INTERVAL` are accepted. Note that if this argument is of list type, the function acts as a `list` function.

Example To get the average salary for each department:

```
SELECT d.name, AVG (d.employees.salary) FROM Department d;
```

COUNT

Syntax `COUNT ([class.|alias.]navigation)`

Purpose Returns the number of successor objects accessible through the relationship navigation.

Example For instance, the following SELECT statement returns each department name and the total number of employees in each department:

```
SELECT d.name, COUNT(d.employees) FROM Department d;
```

MAX

Syntax `MAX ([class.|alias.]navigation.attribute)`

Purpose Returns the maximum value for `attribute` from the successor objects accessible through the relationship navigation.

Argument *`attribute`*

Numeric types, `DATE`, `TIMESTAMP`, and `INTERVAL` are accepted. Note that if this argument is of list type, the function acts as a `list` function.

Example The following SELECT statement returns each department name and the highest salary in the department:

```
SELECT d.name, MAX(d.employees.salary) FROM Department d;
```

MIN

Syntax `MIN ([class.|alias.]navigation.attribute)`

Purpose Returns the minimum value for `attribute` from the successor objects accessible through the relationship navigation.

Argument *`attribute`*

Numeric types, `DATE`, `TIMESTAMP`, and `INTERVAL` are accepted. Note that if this argument is of list type, the function acts as a `list` function.

Example The following SELECT statement returns each department name and the lowest salary in the department:

```
SELECT d.name, MIN(d.employees.salary) FROM Department d;
```

SUM

Syntax `SUM ([class.|alias.]navigation.attribute)`

Purpose Returns the sum value for `attribute` from the set of successor objects that are accessible through the relationship navigation.

Argument *`attribute`*

Numeric types and `INTERVAL` are accepted. Note that if this argument is of list type, the function acts as a `list` function.

10.6 Datetime Functions

This section explains the following datetime functions.

- CURRENT_DATE
- CURRENT_TIMESTAMP
- EXTRACT

CURRENT_DATE

Syntax `CURRENT_DATE()`

Synonyms `CURRENT_DATE`
`CURDATE()`

Purpose Returns the current date in the Universal Coordinated Time zone.

CURRENT_TIMESTAMP

Syntax
```
CURRENT_TIMESTAMP()
```

Synonyms
```
CURRENT_TIMESTAMP
NOW()
```

Purpose Returns the current timestamp in Universal Coordinated Time zone, UTC.

EXTRACT

Syntax
```
EXTRACT(<datetime_field> FROM <value>)
<datetime_field> ::=
    YEAR
  | MONTH
  | DAY
  | HOUR
  | MINUTE
  | SECOND
  | MICROSECOND
<value> ::=
    timestamp value
  | date value
  | interval value
```

Purpose Returns the specified datetime field from a timestamp, date, or interval value. When extracting from a timestamp value, the value returned is in UTC (Universal Coordinated Time) time zone.

Note that when extracting from a date value, only YEAR, MONTH, DAY can be used as <datetime_field>. When extracting from an interval value, DAY, HOUR, MINUTE, SECOND, MICROSECOND can be used as <datetime_field>.

Example The following example extracts the month field from a date value:

```
SELECT EXTRACT (MONTH FROM DATE '1999-11-10') FROM ...;
------------
          11
```

10.7 Conversion Functions

This section describes the following conversion function:

- CAST

CAST

Syntax `CAST (value AS targetType)`

Purpose *For built-in data types:*

CAST converts a value of built-in data type into another built-in data type. *Table 10.1* shows which built-in data types can be converted to which other built-in data types, where the first column represents the source data type and the data types at the top represent the target data types.

Table 10.1 Supported casts between built-in data types

	to STRING	Number types	DATE	TIMESTAMP	INTERVAL	BOOLEAN	CHARACTER	TEXT
from STRING	x	x	x[a]	x[a]	x[a]	x[a]	x[b]	x
Number types	x	x[c]					x[d]	
DATE	x		x					
TIMESTAMP	x		x	x				
INTERVAL	x				x			
BOOLEAN	x					x		
CHARACTER	x	x[d]					x	
TEXT	x							x

CAST does not support any list types. If cast is not supported, the INVALID_CAST error is returned.

(a) When the source type is STRING, string formats for each target type are:

DATE	'yyyy-mm-dd'
TIMESTAMP	'yyyy-mm-dd hh:MM:ss[.uuuuuu]'
INTERVAL	'[+|-]d hh:MM:ss[.uuuuuu]'
BOOLEAN	'TRUE' or 'FALSE' (case insensitive)

If the source string cannot be converted because of incorrect format, INVALID_CAST error is returned.

(b) When the source type is STRING and the target type is CHARACTER, the first character in the source string is returned.

(c) If a source string or a source number value is too big to be represented as the target number type, NUMERICOVERFLOW error is returned.

(d) The conversion between a number and a character is based on ASCII code, i.e., a number is converted into a character whose ASCII value is equivalent to the source number, and vice versa.

When the source value is NULL, CAST returns NULL.

Example

The following example casts a string into a date:

```
SELECT CAST ('1999-11-10' AS DATE) FROM ...
```

The next example normalizes the results of arithmetic division operation into a specific precision and scale:

```
SELECT CAST (num1/num2 AS NUMERIC(19, 4)) FROM ...;
```

Note that if the precision and the scale of the target NUMERIC type are not specified, the default precision and scale (19, 2) are used.

The next example converts a character into an integer:

```
SELECT CAST(CAST('a' AS CHARACTER) AS INTEGER) FROM ...;
----------
        97
```

Note that we need to cast 'a' to the character type since there is no literal expression for a single character.

The next example returns the NUMERICOVERFLOW error, because '123456789' is too big for SHORT type, which ranges from -32768 to 32767.

```
SELECT CAST ('123456789' AS SHORT) FROM ...;  -- Error!!
```

10.8 Numeric Functions

- BIT_COUNT
- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- FLOOR

- LN
- LOG10
- LOG2
- LOG
- MOD
- PI
- POWER
- RADIANS
- ROUND
- SIGN
- SIN
- SQRT
- TAN
- TRUNCATE

BIT_COUNT

Syntax `BIT_COUNT (number)`

Purpose Return the number of bits that are set in the argument *number*.

ABS

Syntax `ABS (number)`

Purpose Returns the absolute value of *number*.

ACOS

Syntax `ACOS (number)`

Purpose Returns the arc cosine of *number*, that is, the value whose cosine is *number*. Returns NULL if *number* is not in the range -1 to 1.

ASIN

Syntax `ASIN (number)`

Purpose Returns the arc sine of *number*, that is, the value whose sine is *number*. Returns NULL if *number* is not in the range -1 to 1.

ATAN

Syntax `ATAN (number)`

Purpose Returns the arc tangent of *number*, that is, the value whose tangent is *number*.

ATAN2

Syntax `ATAN2 (Y,X)`

Purpose Returns the arc tangent of the two variables *X* and *Y*. It is similar to calculating the arc tangent of *Y* / *X*, except that the signs of both arguments are used to determine the quadrant of the result.

CEILING

Syntax `CEILING (number)`

Purpose Returns the smallest integer value not less than *number*.

COS

Syntax `COS (number)`

Purpose Returns the cosine of *number*, where *number* is given in radians.

COT

Syntax `COT (number)`

Purpose Returns the cotangent of *number*.

DEGREES

Syntax `DEGREES (number)`

Purpose Returns the argument *number*, converted from radians to degrees.

EXP

Syntax `EXP (number)`

Purpose Returns the value of e (the base of natural logarithms) raised to the power of *number*. The inverse of this function is LN().

FLOOR

Syntax `FLOOR (number)`

Purpose Returns the largest integer value not greater than *number*.

LN

Syntax `LN (number)`

Purpose Returns the natural logarithm of *number*; that is, the base-e logarithm of *number*. If *number* is less than or equal to 0, then NULL is returned.

LOG10

Syntax `LOG10 (number)`

Purpose Returns the base-10 logarithm of *number*.

LOG2

Syntax `LOG2 (number)`

Purpose Returns the base-2 logarithm of *number*.

LOG

Syntax `LOG (B, X)`

Purpose Returns the logarithm of *X* to the base *B*. If *X* is less than or equal to 0, or if *B* is less than or equal to 1, then NULL is returned.

MOD

Syntax `MOD(m, n)`

Purpose MOD returns the remainder of *m* divided by *n*. Returns *m* if *n* is 0.

The function takes only integer types (i.e., `BYTE`, `SHORT`, `INTEGER`, and `LONG`) as its parameters. The result type is the type of the divisor *n*. The result is negative only if *m* is negative.

PI

Syntax `PI ()`

Purpose Returns the value of pi.

POWER

Syntax `POWER (X, Y)`

Purpose Returns the value of *X* raised to the power of *Y*.

RADIANS

Syntax `RADIANS (number)`

Purpose Returns the argument *number* converted from degrees to radians.

ROUND

Syntax `ROUND (X[, D])`

Purpose Rounds the argument *X* to *D* decimal places. The rounding algorithm depends on the data type of *X*. *D* defaults to 0 if not specified. *D* can be negative to cause *D* digits left of the decimal point of the value *X* to become zero.

SIGN

Syntax `SIGN (number)`

Purpose Returns the sign of the argument *number* as -1, 0, or 1, depending on whether *number* is negative, zero, or positive.

SIN

Syntax `SIN (number)`

Purpose Returns the sine of *number*, where *number* is given in radians.

SQRT

Syntax `SQRT (number)`

Purpose Returns the square root of a non negative number *number*.

TAN

Syntax `TAN (number)`

Purpose Returns the tangent of *number*, where *number* is given in radians.

TRUNCATE

Syntax `TRUNCATE (X, D)`

Purpose Returns the number *X*, truncated to *D* decimal places. If *D* is 0, the result has no decimal point or fractional part. *D* can be negative to cause *D* digits left of the decimal point of the value *X* to become zero.

11 Defining a Schema

This section explains the SQL statements that are used to define a database schema, that is, those that define namespaces, classes, attributes, relationships, indices, entry-point dictionaries, and methods. These statements are called Data Definition Language (DDL). DDL allows you to create, alter, or drop schema objects.

11.1 Namespaces

The `CREATE NAMESPACE` statement allows you to define a namespace into which classes, indexes and entry-point dictionaries can be defined. To modify the namespace definition, you can use the `ALTER NAMESPACE` statement. To remove a namespace from the database, use the `DROP NAMESPACE` statement.

CREATE

Syntax

```
CREATE NAMESPACE [IF [NOT] EXISTS [schema_object]]
nsname[.subnsname]

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Creating Namespace

To create a namespace in the database, you can use the `CREATE NAMESPACE` statement. The following statements create the `com.matisse.example` namespace hierarchy:

```
CREATE NAMESPACE com;
CREATE NAMESPACE com.matisse;
CREATE IF NOT EXISTS NAMESPACE com.matisse.example;
```

ALTER

Syntax

```
ALTER NAMESPACE [IF [NOT] EXISTS [schema_object]]
nsname[.subnsname] RENAME TO new_nsname

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Renaming Namespace

To rename an existing namespace, you can use `ALTER NAMESPACE RENAME` statement. For example, the following statement modifies the `example` sub-namespace name:

```
ALTER NAMESPACE IF EXISTS com.matisse.example
    RENAME TO examples;
```

DROP

Syntax

```
DROP NAMESPACE [IF [NOT] EXISTS [schema_object]]
nsname[.subnsname]

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Dropping Namespace

To remove a namespace from the database, you can use the `DROP NAMESPACE` statement. The following statement removes the `examples` sub-namespace:

```
DROP NAMESPACE IF EXISTS com.matisse.examples;
```

NOTE: a DROP NAMESPACE statement does not delete the sub-namespaces, classes and other schema objects defined inside the namespace. All schema objects inside the namespace are moved into the root namespace.

CURRENT_NAMESPACE

Syntax

```
SET CURRENT_NAMESPACE { DEFAULT | nsname[.subnsname] }
```

Renaming Namespace

This option sets the default namespace where to find schema objects unless their names are fully qualified. DEFAULT refers to the root namespace. For example, the following statement sets the default namespace com.matisse.example.media to where schema objects can be manipulated without their full qualified name:

```
SET CURRENT_NAMESPACE com.matisse.example.media;
CREATE CLASS movie ( ... );
```

11.2 Classes, Attributes, and Relationships

The CREATE CLASS statement allows you to define a class with attributes and relationships. To modify the class definition, you can use the ALTER CLASS statement. It allows you to add, remove, or modify an attribute or relationship. To remove a class from the database, use the DROP CLASS statement.

CREATE

Syntax

```
CREATE {CLASS | TABLE} [IF [NOT] EXISTS [<schema_object>]]
class
  [{UNDER | INHERIT} superclass [, ...]]
(
  <property> [, ...]
  <class_constraint> [, ...]
)

<schema_object> ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
```

```
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)

<property> ::=
    <attribute_name> <attribute_type>
       [DEFAULT <literal>] [NOT NULL] |
    <relationship_name> [READONLY]
      REFERENCES [LIST | SET] (<successor_class>)
      [CARDINALITY (min, max)]
      [INVERSE inv_class.inverse_relationship }]

<class_constraint> ::=
    <unique_constraint> |
    <referential_constraint>

<unique_constraint> ::=
    [CONSTRAINT <name>] {UNIQUE | PRIMARY KEY}
       (<attribute_name> [, ...])

<referential_constraint> ::=
    [CONSTRAINT <name>] FOREIGN KEY (<attribute_name> [, ...])
      REFERENCES <referenced_class> (<attribute_name> [, ...])

<attribute_type> ::=
       AUDIO [(<max>)]|
       IMAGE [(<max>)]|
       VIDEO [(<max>)]|
       TEXT [(<max>)] [<char_code>]|
       CLOB [(<max>)] [<char_code>]|
       BOOLEAN |
       BYTE | TINYINT
       SHORT | SMALLINT
       INTEGER | INT
       LONG | BIGINT
       NUMERIC [(<precision>[,<scale>])]|
       FLOAT | REAL
       DOUBLE [PRECISION]|
       CHAR [(<n>)]| CHARACTER |
```

```
STRING [<char_code>]|
VARCHAR [(<n>)][<char_code>]|
NVARCHAR [(<n>)] |
DATE |
TIMESTAMP |
INTERVAL |
BYTES [(<max>)] |
LIST(BOOLEAN [, <max>]) |
LIST(SHORT [, <max>]) |
LIST(INTEGER [, <max>]) |
LIST(LONG [, <max>])|
LIST(NUMERIC [(<precision>[,<scale>] [, <max>]))|
LIST(FLOAT [, <max>])|
LIST(DOUBLE [, <max>])|
LIST(STRING [, <max>])|
LIST(DATE [, <max>])|
LIST(TIMESTAMP [, <max>])|
LIST(INTERVAL [, <max>])
```

```
<char_code> ::=
        CHARACTER SET (UTF8 | UTF16)
```

`<literal>`: See *section 2.1, What Is a Constant?*

`<successor_class>`: Class as a successor type for relationship.

`<inv_class>`: Class where `inverse_relationship` is defined.

Inheritance Class inheritance can be specified using the keyword `UNDER` or `INHERIT`. For example, to define the `movieDirector` class as a subclass of the `artist` class,

```
CREATE CLASS movieDirector INHERIT artist (
   ...
);
```

Matisse supports multiple inheritance. The `INHERIT` clause can have more than one class. For example, to define the `movieDirector` class as a subclass of both the `artist` class and the `director` class,

```
CREATE CLASS movieDirector INHERIT artist, director (
   ...
);
```

The `INHERIT` clause is optional.

Attribute An attribute is defined with its name, type and an optional default value. Possible types are shown above in the syntax. An attribute can accept only one type, or a `NULL` value unless the `NOT NULL` keyword is specified. For example, the values for the attribute `synopsis` in the following example can be `STRING` or `NULL` type:

```
CREATE CLASS movie (
    synopsis STRING,
    ...
);
```

while the attribute `title` in the following example can be only `STRING` type:

```
CREATE CLASS movie (
    title STRING NOT NULL,
    ...
);
```

An attribute may have a default value. For example,

```
CREATE CLASS movie (
    category STRING DEFAULT 'non genre',
    ...
);
```

If you do not set a value for the category attribute in a `movie` object, the object will have the string “non genre” for the attribute as default value.

For more information about constant literal, please refer to *section 2.1, What Is a Constant?*.

Note that the attributes defined in this syntax are local to the class.

Note that attribute definitions and relationships definition can appear in a class definition in any order.

Maximum Size of Attribute

With the type `VARCHAR(n)`, you can set the maximum length of characters to n. The maximum length needs to be between 1 and 2147483648 (2G). The default maximum length is 2G.

With the media types like `AUDIO(n)` or `BYTES(n)`, you can set the maximum size in bytes. The maximum size needs to be between 1 and 2147483648 (2G). The default maximum size is 2G. The maximum size can be specified like 10K or 20M for 10 kilo-bytes or 20 mega-bytes respectively.

```
CREATE CLASS movie (
    title VARCHAR(100),
    preview VIDEO(5M),
    ...
);
```

NOTE: For ODBC: This maximum length or size is returned as the maximum column size for these types. If the maximum length of size is not specified, 2147483648 is returned.

For the list types, you can optionally specify the maximum number of elements in a list with the following syntax, e.g.,:

```
LIST(INTEGER, 20)
```

Relationship A relationship is defined with its name, a class of successor objects, an optional inverse relationship and optional cardinality numbers. For example, the next statement defines a relationship `playingMovies` for the class `Theater`:

```
CREATE CLASS Theater (
  playingMovies REFERENCES (movie)
);
```

The next examples define a relationship `directedBy` whose successor class is `movieDirector` and inverse relationship is `direct` defined in the `movieDirector` class:

```
CREATE CLASS movie (
    directedBy REFERENCES (movieDirector)
        INVERSE movieDirector.direct,
    ...
);
CREATE CLASS movieDirector (
    direct REFERENCES (movie)
        INVERSE movie.directedBy,
    ...
);
```

In the above example, the cardinality numbers for the relationship are not provided. The default values for the minimum relationship cardinality is 0 and the maximum one is unlimited. The cardinality definition in the following statement is same as the default:

```
CREATE CLASS movie (
    directedBy REFERENCES (movieDirector)
        CARDINALITY (0, -1)
        INVERSE movieDirector.direct,
    ...
);
```

To let a single movie director direct a movie, the relationship cardinality should be (1, 1), or (0, 1) in which case a movie does not necessarily have to have a director. For example,

```
CREATE CLASS movie (
    directedBy REFERENCES (movieDirector)
        CARDINALITY (0, 1)
        INVERSE movieDirector.direct,
    ...
);
```

By default, the successor objects of a relationship is not ordered and does not keep their order as you add or remove successor objects so that Matisse can store these objects in any order for best performance. However, if the LIST keyword is following REFERENCES, the successor objects do keep their order. For example:

```
CREATE CLASS movie (
    directedBy REFERENCES LIST (movieDirector)
        INVERSE movieDirector.direct,
    ...
);
```

In Matisse, relationships can be given an explicit directional orientation, that is, a regular or an inverse relationship. You cannot manipulate objects through a relationship that is explicitly defined as an inverse relationship. The relationships defined above are not given an explicit directional orientation. You can set a `movieDirector` object through the `directedBy` relationship in a `movie` object, and you also can set `movie` objects through the relationship `direct` in a `movieDirector` object.

To define an inverse relationship explicitly, use the `READONLY` keyword as shown below, for example:

```
CREATE CLASS car (
    wheels REFERENCES (tire)
        INVERSE tire.componentOf,
    ...
);
CREATE CLASS tire (
    componentOf READONLY REFERENCES (car)
        CARDINALITY (0, 1)
        INVERSE car.wheels,
    ...
);
```

This is useful in application development when you want to prohibit defining interfaces that manipulate `car` objects through the `componentOf` relationship in the `tire` class. That is, you can define an interface like `setWheels(tire1, tire2, ...)` or `replaceWheel(tire, position)` in the `car` class but you cannot define an interface like `detachFrom(car)` in the `tire` class.

Note again that attribute definitions and relationship definitions can appear in a class definition in any order.

Unique Constraint

A class can contain unique constraints and/or referential constraints. Unique constraint enforces the uniqueness of values of an attribute or a set of up to four attributes. The attributes used for unique constraint cannot be nullable, i.e., they need to be `NOT NULL`.

For example, if you want to make the pair of movie title and its category unique:

```
CREATE CLASS movie (
  title VARCHAR(150) NOT NULL,
  category VARCHAR(50) NOT NULL,
  CONSTRAINT unique_title_category UNIQUE (title, category)
);
```

A unique constraint can use up to 256 characters, thus the sum of the sizes for the attributes should not exceed 256.

Note that using `PRIMARY KEY` instead of `UNIQUE` has the same effect for unique constraint.

Referential Constraint

The referential constraint is provided for the purposes of compatibility with relational database products. It generates a relationship (and its inverse relationship) between the class (table) and the referenced class (table).

For example, the following two statements will generate a relationship `Companies_companyId` and its inverse relationship `to_Persons_companyId` between class `Persons` and class `Companies`:

```
CREATE TABLE Companies (
  companyId VARCHAR(20) NOT NULL,
  CONSTRAINT companyId_pk PRIMARY KEY (companyId)
);

CREATE TABLE Persons (
  personId VARCHAR(20) NOT NULL,
  companyId VARCHAR(20),
  CONSTRAINT workFor_fk FOREIGN KEY (companyId)
      REFERENCES Companies (companyId)
);
```

Note that the referential constraint in Matisse is not provided to define a relationship between classes, but to make it possible to run the SQL DDL statements written for relational database products.

ALTER

Syntax

```
ALTER {CLASS | TABLE} [IF [NOT] EXISTS [schema_object]]
class
         DROP { ATTRIBUTE attribute
              | REFERENCES relationship
              | {INHERIT | UNDER} <superclass>
              }
```

```
ALTER {CLASS | TABLE} class
          ADD { ATTRIBUTE attribute attribute_type
                  [DEFAULT literal] [NOT NULL]
              | RELATIONSHIP relationship
                  [[READONLY] REFERENCES [LIST | SET]]
                  ( succ_class [, ...] )
                  [CARDINALITY (min, max)]
                  [INVERSE inv_class.inverse_relationship]
              | {INHERIT | UNDER} <superclass>
              }
ALTER {CLASS | TABLE} class
          ALTER { ATTRIBUTE attribute attribute_type
                    [DEFAULT literal] [NOT NULL]
                | RELATIONSHIP relationship
                    [[READONLY] REFERENCES [LIST | SET]]
                    ( succ_class [, ...] )
                    [CARDINALITY (min, max)]
                    [INVERSE inv_class.inverse_relationship]
                }
ALTER {CLASS | TABLE} class
          RENAME { TO new_class
                  | ATTRIBUTE attribute TO new_attribute
                 | RELATIONSHIP relationship
                 }

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

`attribute_type`: See *CREATE, on page 117*.
`literal`: See *section 2.1, What Is a Constant?*.
`succ_class`: Class as a successor type for relationship.
`inv_class`: Class where `inverse_relationship` is defined.

Drop Properties

To drop an attribute, relationship, or superclass in a class, you can use `ALTER CLASS DROP` statement. For example, the following statement drops the `synopsis` attribute from the `movie` class:

```
ALTER CLASS movie DROP ATTRIBUTE synopsis;
```

Add Properties

To add a new attribute, relationship, or superclass in a class, you can use `ALTER CLASS ADD` statement. For example, the following statement adds a new attribute `releasedDate` to the `movie` class:

```
ALTER CLASS movie
    ADD ATTRIBUTE releasedDate DATE;
```

The following example adds a new relationship, `starring`, to the `movie` class:

```
ALTER CLASS movie
    ADD RELATIONSHIP starring REFERENCES (artist)
       INVERSE artist.biography;
```

The following example adds a new superclass, `artist`, to the `movieDirector` class:

```
ALTER CLASS movieDirector
    ADD INHERIT artist;
```

Modify Properties

To modify an existing attribute or relationship in a class, you can use `ALTER CLASS ALTER` statement. For example, the following statement modifies the `category` attribute in the `movie` class so that every `movie` object must have some category name:

```
ALTER CLASS movie
    ALTER ATTRIBUTE category STRING NOT NULL;
```

The following example modifies the `starring` relationship defined above so that it can have 10 starrings at most:

```
ALTER CLASS movie
    ALTER RELATIONSHIP starring REFERENCES (artist)
       CARDINALITY (0, 10)
       INVERSE artist.biography;
```

Rename Properties

To rename an existing class or an existing attribute or relationship in a class, you can use `ALTER CLASS RENAME` statement. For example, the following statement modifies the `movie` class name:

```
ALTER CLASS movie
    RENAME TO movies;
```

The following example renames the `category` attribute to `categories`:

```
ALTER CLASS movie
    RENAME ATTRIBUTE category TO categories;
```

DROP

Syntax

```
DROP {CLASS | TABLE} [IF [NOT] EXISTS [schema_object]] class

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Dropping Class

To remove a class from the database, you can use the `DROP CLASS` statement. The following statement removes the `movie` class:

```
DROP CLASS IF EXISTS movie;
```

Note that a `DROP CLASS` statement removes the attributes and relationships defined in the class unless they are used by other classes.

11.3 Indexes

Matisse provides a conventional indexing mechanism, which allows you to index objects of a class using up to four attributes. You can look up objects by value intervals. A Matisse index can be created or deleted at any time without interrupting concurrent operations.

CREATE

Syntax

```
CREATE [UNIQUE] INDEX [IF [NOT] EXISTS [schema_object]]
index ON class (
    attribute [ASC | DESC]
    [, ...]
)

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
```

```
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Criteria

An index can have four attributes as its criteria at most. They must be defined in the class on which you are going to create the index. Each criterion attribute may specify a direction, ascending or descending, in which objects are to be indexed. This is optional and the default direction is ascending.

The total size occupied by all the attributes to be indexed must not exceed 256 bytes.

If the optional UNIQUE keyword is specified, each entry in the index needs to be unique, allowing them to be used as primary keys. By default, an index is defined as non-unique.

The following example shows how to create an index on the `movie` class using the two attributes `title` and `runningTime`:

```
CREATE CLASS movie (
    title VARCHAR(150) NOT NULL,
    runningTime INTEGER NOT NULL
);
CREATE INDEX movieIndex ON movie (
    title ASC,
    runningTime ASC
);
```

DROP

Syntax

```
DROP INDEX [IF [NOT] EXISTS [schema_object]] index

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
```

```
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Dropping Index

To remove an index, you can use `DROP INDEX` statement. The following statement removes the index `movieIndex`:

```
DROP INDEX IF EXISTS movieIndex;
```

11.4 Entry Point Dictionaries

Matisse provides another indexing mechanism called entry point dictionary. An entry point dictionary indexes objects by keywords, also called entry points. You can then retrieve the objects via their entry points.

CREATE

Syntax

```
CREATE [UNIQUE] ENTRY_POINT DICTIONARY [IF [NOT] EXISTS
[schema_object]]
  entry_point_dictionary_name
  ON class (attribute)
  [CASE SENSITIVE]
  [MAKE_ENTRY make_entry_function]

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Make-Entry Function

An entry-point dictionary is defined on an attribute with an entry-point function. An entry-point function generates an entry-point string for an object, which is used to index the object. The default value for `make_entry_function` is "`make-entry`". The alternative value is "`make-full-text-entry`", which generates entry-point strings for every word contained in a character string attribute.

If the optional CASE SENSITIVE is specified, entry point dictionary lookups are case sensitive. By default, the lookups are case insensitive.

If the optional UNIQUE keyword is specified, each entry in the entry-point dictionary needs to be unique. By default, an entry-point dictionary is defined as non-unique

The following example defines an entry-point dictionary `titleDict` on the `title` attribute with the `make-full-text-entry` make-entry function:

```
CREATE ENTRY_POINT DICTIONARY titleDict ON movie(title)
    MAKE_ENTRY "make-full-text-entry";
```

Note that an entry-point function can generate several entry-point strings for an object. For more details about entry point dictionary, please refer to the *Getting Started with Matisse*.

DROP

Syntax

```
DROP ENTRY_POINT DICTIONARY [IF [NOT] EXISTS
[schema_object]] entry_point_dictionary_name

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Removing Entry Point Dictionary

To remove an entry point dictionary, you can use the `DROP ENTRY_POINT DICTIONARY` statement. The following example removes the entry point dictionary `titleDict` defined for the `title` attribute in the `movie` class:

```
DROP ENTRY_POINT DICTIONARY titleDict;
```

11.5 Methods

Matisse supports SQL Methods, as defined in the SQL-99 standard, enabling you to define and store programs written in SQL. This provides you a full fledged object-oriented programming capability in the database server, thus giving you faster execution, better reuse of code and maintenance.

CREATE

Syntax

```
CREATE [INSTANCE | STATIC] METHOD [IF [NOT] EXISTS
[schema_object]] method_name (<parameter_declaration [,
...]>)
RETURNS <data_type>
FOR class_name
BEGIN
  <statement>;
  [...]
END

schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)

<parameter_declaration> ::=
  [ IN ] parameter_name <data_type>
<data_type> ::=
  <attribute_type> |
  <object_type>
```

Creating a New Method

This DDL statement creates a new method, and stores it in the database as an instance of the meta-schema class MtMethod.

In this release of Matisse, `<parameter_declaration>` supports only input parameter, specified by the `IN` keyword. The other two types, `OUT` (output parameter) and `INOUT` (both for input and output), will be supported in a next release.

The data a method can return is either of attribute type, for example `INTEGER` or `DATE`, or of object type such as class `Movie`, or of object list type such as class `Movie` selection, or of type `table`. If a method does not return anything, its return type is `NULL`.

Static Method

`CREATE STATIC METHOD` statement creates a static method, which belongs to a class specified after `FOR` and does not operate on each instance of a class, but can be used with `CALL` statement.

For example, a method returning an `INTEGER`:

```
CREATE STATIC METHOD count_movie(a_pattern STRING)
RETURNS INTEGER
FOR movie
BEGIN
  DECLARE cnt INTEGER;
  SELECT COUNT(*) INTO cnt FROM movie
   WHERE title LIKE a_pattern;
  RETURN cnt;
END;

CALL movie::count_movie ('R%');
```

For example, a method returning an object selection:

```
CREATE STATIC METHOD listPresidentsBetween(startYear INT,
endYear INT)
  RETURNS SELECTION(Person)
  FOR Presidency
  BEGIN
    DECLARE sel SELECTION(Person);
    SELECT REF(p) FROM Person p
      WHERE p.IsInChargeOf.EndingYear >= startYear
            AND p.IsInChargeOf.StartingYear <= endYear
      INTO sel;
    RETURN sel;
  END;
```

For example, a method returning a table containing scalar values:

```
CREATE STATIC METHOD viewPresidentsBetween(startYear INT,
endYear INT)
  RETURNS TABLE(firstName VARCHAR(32), lastName
VARCHAR(32), startingYear INT, endingYear INT)
  FOR Presidency
  BEGIN
    SELECT p.FirstName, p.LastName,
p.IsInChargeOf.StartingYear, p.IsInChargeOf.EndingYear
FROM Person p
      WHERE p.IsInChargeOf.EndingYear >= startYear
            AND p.IsInChargeOf.StartingYear <= endYear;
  END;
```

For example, a method returning a table containing objects:

```
CREATE STATIC METHOD LocateEmployees(aCity String,
aMinSalary INT, aMaxSalary INT)
RETURNS TABLE(city STRING, department STRING, emp Employee)
FOR Employee
```

```
BEGIN
SELECT FILTERED
   e.Address.City,
   e.Department.DepartmentName,
   Ref(e)
FROM
   Employee e
WHERE
   e.Address.City = aCity
   AND e.Salary between aMinSalary and aMaxSalary
ORDER BY
   e.Department.DepartmentName,
   e.Salary;
END;
```

Updating a Method

Use `CREATE METHOD` statement to update an existing method. The statement updates the definition of a method, if the specified method already exists in the database.

NOTE: Execute 'COMPILE ALL' after any changes to the database schema including methods, so that all the methods are valid with the latest schema.

DROP

Syntax

```
DROP METHOD [IF [NOT] EXISTS [schema_object]] method_name
FOR class_name
```

```
schema_object ::=
  SCHEMA_OBJECT(NAMESPACE,[<ns path name>.]<ns name>)
| SCHEMA_OBJECT(CLASS,[<ns path name>.]<class name>)
| SCHEMA_OBJECT(ATTRIBUTE,[<ns path name>.]<cls name>.<att
name>)
| SCHEMA_OBJECT(RELATIONSHIP,[<ns path name>.]<cls
name>.<rel name>)
| SCHEMA_OBJECT(METHOD,[<ns path name>.]<cls name>.<mth
name>)
| SCHEMA_OBJECT(INDEX,[<ns path name>.]<index name>)
| SCHEMA_OBJECT(ENTRY_POINT DICTIONARY,[<ns path
name>.]<entry point name>)
```

Removing Method

A `DROP METHOD` statement removes a method defined for `class_name` from the database. For example:

```
DROP METHOD IF EXISTS count_movie FOR movie;
```

COMPILE

Syntax

```
COMPILE METHOD method_name FOR class_name;
```

```
COMPILE ALL;
```

Recompile Methods

When you create new methods or update methods using `CREATE METHOD` statement, the methods are compiled and stored in the database.

However, as you update the database schema, for example removing an attribute, adding a new class, or updating methods, some methods could become inconsistent with the schema, since Matisse does not recompile all the methods automatically after any changes of schema. You need to run COMPILE statement to make methods consistent with schema before executing methods.

The `COMPILE METHOD` statement compiles a specific method, while the `COMPILE ALL` statement compiles all the methods stored in the database. It's safe to use `COMPILE ALL` when you update the database schema.

12 Manipulating Data

This section explains how to perform the following functions with SQL:

- Insert new objects into a database
- Update attributes or relationships of objects
- Delete some objects

These statements need to be executed within a transaction, not a version access.

12.1 Inserting Data

INSERT

An `INSERT` statement creates a new object of a given class, and sets its attribute values and relationship successors.

Syntax

```
INSERT INTO class
    [(properties_list)]
    VALUES (property_values_list)
    [returning_clause]
properties_list
    ::= attribute_or_relationship [, ...]
property_values_list
    ::= expression [, ...]
returning_clause
    ::= RETURNING [REF(class)] INTO a_selection
```

Attributes

You can set a literal constant as a new value for an attribute, For example, the following statement creates a new instance of the `artist` class:

```
INSERT INTO artist
    (lastName, firstName)
    VALUES ('Roberts', 'Julia');
```

The next example creates an instance of the `boxOffice` class:

```
INSERT INTO boxOffice
  (week, topReceipts)
  VALUES (DATE '2001-01-29',
          LIST(NUMERIC(10, 2))(34.5, 20.0, 8.9, 3.3, 2.1));
```

In this example, the new `boxOffice` object will have a default value of 0 for the attribute `totalReceipts`, since its value is not provided in the statement, and the attribute `totalReceipts` is defined with this default value. If the attribute does not have a default value and it allows a `NULL` value, then the attribute value for the object remains unspecified.

Relationships

You can set a list of objects for a relationship in an `INSERT` statement. The following example creates a `movie` object for the movie *Erin Brockovich* with Julia Roberts starring.

```
SELECT REF(a) FROM artist a
    WHERE a.lastName = 'Roberts' and a.firstName = 'Julia'
    INTO anActress;
INSERT INTO movie
    (title, category, rating, ... , starring )
    VALUES ('Erin Brockovich', 'Drama', 'R', ..., anActress);
```

As a value for a relationship, you can use a selection, the selection constructor `SELECTION`, or set operations on selections as shown in the previous selection Updating Data.

Returning clause

An `INSERT` statement with a returning clause retrieves the object created and stores it in a selection. This selection can then be used in other SQL statements until it is freed.

The following example creates an `artist` object and store it in a selection, then creates a `movie` object using the selection:

```
INSERT INTO artist (firstName, lastName)
    VALUES ('Tom', 'Cruise')
    RETURNING REF(artist) INTO aSelection;

INSERT INTO movie (title, starring)
    VALUES ('Minority Report', aSelection);
```

12.2 Updating Data

UPDATE

You can update objects with the `UPDATE` command. The command updates attribute values or relationship successors of the objects that qualify the predicate of an SQL statement. The command returns the number of objects updated.

Syntax

```
UPDATE class SET
    attribute = { expression | DEFAULT | NULL } | [, ...]
```

```
relationship = expression [, ...]
[WHERE search_condition]
```

In this syntax, `attribute` is an attribute name, `relationship` is a relationship name, `expression` is a value or an object selection to be set, and `search_condition` is a predicate to select objects.

Attributes

As a new value for an attribute, you can set a literal constant. For example, the following statement updates the rank of the movie *Thirteen Days* for a week:

```
UPDATE movie SET rankForWeek = 5
    WHERE title = 'Thirteen Days';
```

You can also set an attribute to its default value. For example, the following statement updates the movie weekly ranking to its default value (i.e. no rank):

```
UPDATE movie SET rankForWeek = DEFAULT
    WHERE rankForWeek IS NOT DEFAULT;
```

Relationships

You can add, remove, or replace successor objects through a relationship using the `UPDATE` statement. There are several ways to manipulate successor objects.

1. Using a selection

   A selection is a set of objects created by a `SELECT INTO` query. If you create a selection of objects using the `INTO` clause, you can then use it to set successor objects for a relationship. In the following statements, the first one selects the top 10 movies of a week and saves the result into a selection. The second statement then assigns the selection to the `topTitles` relationship in a `boxOffice` object.

   ```
   SELECT REF(m) FROM movie m
       WHERE m.rankForWeek >= 1 AND m.rankForWeek <= 10
       ORDER BY m.rankForWeek
       INTO top10Titles;
   UPDATE boxOffice
       SET topTitles = top10Titles
       WHERE week = DATE '2001-01-22';
   ```

2. Using the selection constructor `SELECTION`

   The `SELECTION` keyword constructs a new selection using relationships, other selections or OIDs (Object Identifiers).

   The following statements show how to append into the `topTitles` relationship other movies whose ranks are between 11 and 20.

   ```
   SELECT REF(m) FROM movie m
       WHERE m.rankForWeek >= 11 AND m.rankForWeek <= 20
       ORDER BY m.rankForWeek
       INTO next10Titles;
   UPDATE boxOffice
       SET topTitles = SELECTION(topTitles, next10Titles)
   ```

```
    WHERE week = DATE '2001-01-22';
```

The SELECTION operation can take more than two arguments, which are either relationship or selection.

```
SELECT REF(m) FROM movie m
    WHERE m.rankForWeek >= 21 AND m.rankForWeek <= 30
    ORDER BY m.rankForWeek
    INTO more10Titles;
UPDATE boxOffice
    SET topTitles =
      SELECTION(top10Titles, next10Titles, more10Titles)
    WHERE week = DATE '2001-01-22';
```

The SELECTION operation can take OIDs as arguments. OIDs can be either decimal or hexadecimal (prefixed by 0x). If you know the OIDs for the top five movie titles, you may write the following statement to update the topTitles relationship:

```
UPDATE boxOffice
    SET topTitles =
      SELECTION('1234', '1236', '1238', '0x4E6', '0x4E8')
    WHERE week = DATE '2001-01-22';
```

3. Empty relationship

   To remove all successor objects of a relationship, you can use the empty selection SELECTION(). The following statement removes all successor objects, if any, for the topTitles relationship:

```
UPDATE boxOffice
    SET topTitles = SELECTION()
    WHERE week = DATE '2001-01-29';
```

4. Set operations on selections

   You can use set operations to set successor objects. Three kinds of set operators for selections are provided: UNION, INTERSECT, and EXCEPT. They take two operands, both of which are selections or another set operation expression.

```
selection1 UNION selection2
```

The UNION operator returns a union of two selections: selection1 and selection2. The order of objects is not preserved.

```
selection1 INTERSECT selection2
```

The INTERSECT operator returns an intersection of two selections: selection1 and selection2. The order of objects is not preserved.

```
selection1 EXCEPT selection2
```

The EXCEPT operator returns a difference of two selection selection1 and selection2, that is, all objects in selection1 except those in selection2. The order of objects is preserved.

The following example shows how to filter selected movies by their ratings:

```
SELECT REF(m) FROM movie m
    WHERE m.rating = 'G' OR m.rating = 'PG'
    INTO kMovies;
UPDATE boxOffice
    SET topTitlesForKids =
     SELECTION(SELECTION(top10Titles, next10Titles)
INTERSECT kMovies)
    WHERE week = DATE '2001-01-22';
```

The second statement above can also be stated as follows:

```
UPDATE boxOffice
    SET topTitlesForKids =
      SELECTION((top10Titles UNION next10Titles) INTERSECT
gMovies)
    WHERE week = DATE '2001-01-22';
```

The following is another example filtering selected movies by their ratings. It is excluding movies rated NC-17.

```
SELECT REF(m) FROM movie m
    WHERE m.rating = 'NC-17'
    INTO ncMovies;
UPDATE boxOffice
    SET topTitles =
      SELECTION(SELECTION(top10Titles, next10Titles) EXCEPT
ncMovies)
    WHERE week = DATE '2001-01-22';
```

12.3 Deleting Data

DELETE

A DELETE statement deletes a set of objects that qualifies the statement's where clause. If the statement does not have a where clause, it deletes all the instances of the class.

Syntax

```
DELETE FROM class [ WHERE search_condition ]
```

Example

The following example deletes all the boxOffice objects whose records are older than Jan. 01, 1985.

```
DELETE FROM boxOffice
    WHERE week < DATE '1985-01-01';
```

12.4 Auto Increment Attribute

Each object in Matisse has a unique object identifier that can be accessed via the OID attribute. If your application requires to define a primary key attribute with an auto incremental value, you can rely on the Matisse OIDs which are unique and incremental to build a unique and incremental key.

Example You can set the GUID attribute value based upon the object unique OID as follows:

```
INSERT INTO movie (GUID, title)
       VALUES(CAST(OID AS INT),'La Vie en Rose');
```

13 Stored Methods and Statement Blocks

Matisse supports stored methods, which are like stored procedures for relational database systems but provide an object-oriented programming environment with inheritance and polymorphic behavior. Matisse stored methods are stored and executed in the database server, and offer several advantages:

1. Performance. Methods are precompiled and stored in the server. They execute much faster than compiling SQL statements upon each execution. Methods usually contain several SQL statements and generate much less network traffic compared to executing each SQL statement from the client one by one.
2. Reusability. A stored method can be used by different client side applications, ensuring that they use the same business logic, and reducing the risk of application programming error.
3. Extensibility. When you extend the application by adding new subclasses, these subclasses can reuse the methods defined in their superclasses, or redefine them to implement the new behavior. This is also known as polymorphism.
4. Maintainability. Well defined methods hide all the details of the data structures. When updating the data structures or database schema, you can minimize the changes to your application with the use of methods.

Matisse stored methods follow the syntax of SQL-99 PSM.

For information about creation, update, and deletion of methods, please refer to *11.5 Methods*.

13.1 A Simple Example

The following example provides a brief overview of Matisse SQL methods. First, we define a method for class `Artist` that returns the actor's full name:

```
CREATE METHOD full_name()
RETURNS STRING
FOR Artist
BEGIN
  RETURN CONCAT(firstName, CONCAT(' ', lastName));
END;
```

Then, we define another method for class `MovieDirector` with the same name, overriding the method defined for `Artist`, since `MovieDirector` is inheriting from `Artist`:

```
CREATE METHOD full_name()
RETURNS STRING
FOR MovieDirector
BEGIN
  -- Put the title 'Director' before the name, and use
  -- only the initial letter for the first name.
  DECLARE firstInitial STRING;
  SET firstInitial = CONCAT(SUBSTR(firstName, 1, 1), '. ');
  RETURN CONCAT ('Director ',
                  CONCAT(firstInitial, lastName));
END;
```

Now, execute a SELECT statement to check if there are actors or movie directors who have more than 20 letters in the full name returned by the full_name() method:

```
SELECT firstName, lastName FROM Artist a
 WHERE LENGTH(a.full_name()) > 20;
firstName            lastName
-------------------- --------------------
Steven               Spielberg
1 objects selected
```

Note that the above SELECT query searches for both `Artist` instances and `MovieDirector` instances, and it invokes both the `full_name()` method which is defined for `Artist` instances and the `full_name()` method which is defined for `MovieDirector` instances.

13.2 Method Invocation

A method can be called within a SELECT statement, another method, or almost anywhere an expression is allowed. The basic form to invoke a method is:

```
object.method(<parameter list>)
```

Calling a Method in SELECT Statement

In a SELECT statement, since an alias name for the class in FROM clause is representing each object in the class, a method can be called as following:

```
SELECT d.full_name()
FROM MovieDirector d
WHERE LENGTH(d.full_name()) < 30;
```

Note that currently methods can be called only in the WHERE clause of SELECT, UPDATE, or DELETE statements.

Calling a Method in Method Body

In a method body, there are several ways to invoke a method.

(1) Using FOR statement

The FOR statement, explained later in this section, takes a loop variable of object type, on which you can call a method. For example,

```
CREATE STATIC METHOD total_length()
RETURNS INTEGER
FOR Artist
BEGIN
  DECLARE len INTEGER;
  FOR obj AS SELECT REF(a) FROM Artist a DO
    SET len = len + LENGTH(obj.full_name());
  END FOR;
  RETURN len;
END;
```

(2) Using SELF

The SELF keyword is a pseudo variable referring to the object on which the method operates. You can invoke a method using SELF, for example,

```
CREATE METHOD full_name_length()
RETURNS INTEGER
FOR Artist
BEGIN
  RETURN LENGTH(SELF.full_name());
END;
```

(3) Method Parameter or Relationship Successor Object

You can pass an object as a method parameter and invoke a method on the object. For example,

```
CREATE METHOD aMethod1(anArtist Artist)
RETURNS INTEGER
FOR Movie
BEGIN
  DECLARE len INTEGER;
  SET len = anArtist.full_name_length();
  ...
END;
```

You can get a successor object from a relationship, and invoke a method on the object. For example,

```
BEGIN
  DECLARE aMovie Movie;
  DECLARE star Artist;

  -- Select a Movie object into a variable
  SELECT REF(m) INTO aMovie FROM Movie m WHERE ...;

  -- Get the first successor object from the Starring
relationship
  SET star = aMovie.Starring(1);
```

```
  RETURN start.full_name();
END;
```

Calling a Method with LOOKUP

An instance method can be called using the `CALL` keyword. An example is shown in the following section.

Syntax

```
CALL LOOKUP(<class-name>,<oid>).<method-name>
([<parameter> [, ...]])
```

Example

Call the instance method `getSalaries()` on the `Employee` instance `'0x1234'`, as a single statement:

```
CALL LOOKUP("Employee",'0x1234').getSalaries(2008);
```

This returns an integer value.

Calling a Static Method

A static method can be called using the `CALL` keyword. Inside the WHERE-clause of SELECT, UPDATE, or DELETE statement, it can be called without using the `CALL` keyword. An example is shown in the following section.

Syntax

```
CALL <class-name>::<method-name> ([<parameter> [, ...]])
```

Example

Call the static method defined above:

```
CALL Artist::total_length();
```

This returns an integer value.

Static Method and Query Optimization

When a static method is used with a query statement, the static method will be executed only once if the method has no correlated reference to the query statement. For example, if we define a simple static method that returns the average running time of all the movies for a given rating:

```
CREATE STATIC METHOD avg_run_time(aRate STRING)
RETURNS DOUBLE
FOR Movie
BEGIN
  DECLARE avgtime DOUBLE;
  SELECT AVG(runningTime) INTO avgtime FROM Movie
   WHERE rating = aRate;
  RETURN avgtime;
END;
```

Then, the next query selects movies rated as ‘PG-13’ and having more running time than average running time for all the movies rated as ‘PG-13’:

```
SELECT *
FROM Movie
WHERE
  rating = 'PG-13'
  AND runningTime > Movie::avg_run_time('PG-13');
```

For this query, the method avg_run_time does not need to be executed for each Movie object, but it is sufficient to run it once. Matisse detects this situation, and optimizes the query so that it executes the static method only once.

Calling a Method in a Superclass

When you need to call a method in a superclass from its subclass's method, you can use the generalized method invocation:

```
(<object expression> AS class_name).method(...)
```

A typical usage of the generalized method invocation is initialization of an object. For example,

```
CREATE METHOD Initialize()
RETURNS NULL
FOR MovieDirector
BEGIN
  -- Call the initialization method in its superclass
  (SELF AS Artist).Initialize();
  -- ... some other initialization here
END;
```

Calling a Method returning a Table

A method can return a table which contains objects. An example is shown in the following section.

Syntax

```
CALL <class-name>::<method-name> ([<parameter> [, ...]])
```

Example

Call the static method defined in the method definition section *11.5 Methods* :

```
CALL Employee::LocateEmployees('Lyon', 95000, 150000);
```

This returns a SQL projection which contains Employee objects.

```
city         department                emp
------------ ------------------------- ----------------
Lyon         Customer Care             0x3f63 Manager
Lyon         Finance                   0x3322 Manager
Lyon         Information Technology    0x5ad2 Employee
Lyon         Sales                     0x1b6f Employee
Lyon         Sales                     0x14c0 Employee
Lyon         Sales                     0x4642 Director
```

13.3 Update Object in a Method

In an instance method (i.e., in CREATE METHOD statement), you can use UPDATE SELF statement to update attributes or relationships of the object on which the method is called.

For instance, the following method receives a string for movie rating as a parameter, and checks if the rating string is valid. If it is valid, it execute an UPDATE statement to update the `Rating` attribute:

```
CREATE METHOD UpdateRating(newRating STRING)
RETURNS NULL
FOR Movie
BEGIN
  IF newRating IN ('G', 'PG', 'PG-13', 'R') THEN
    UPDATE SELF SET Rating = newRating;
  ELSE
    ...
  END IF;
END;
```

Note that UPDATE SELF statement is not allowed in a static method (i.e., CREATE STATIC METHOD statement).

13.4 Control Statements

Control statements control the flow of the program, the declaration and assignment of variables, and handles exceptions, which are allowed to be used in a method body or a statement block. Control statements allow you to write a program in a way writing programs in complete programming languages.

Matisse provides the following control statements:

- IF
- LOOP
- REPEAT
- WHILE
- FOR
- LEAVE
- ITERATE
- RETURN
- SET assignment
- SIGNAL
- RESIGNAL

IF Statement

The IF statement evaluates a condition and selects a different execution path depending on the result.

Syntax

```
IF <condition> THEN
  <list of statements>
[ ELSEIF <condition> THEN
    <list of statements> ]
[ ELSE
    <list of statements> ]
END IF;
```

If `<condition>` evaluates to true, then the following `<list of statements>` will be executed. Otherwise, it tries the next `<condition>`, and if it is true, the following `<list of statements>` will be executed.

If no `<condition>` evaluates to true and ELSE clause is provided, `<list of statements>` in ELSE clause is executed.

Example The following method returns the absolute value of an integer:

```
CREATE METHOD abs (arg INTEGER)
...
BEGIN
  IF arg < 0 THEN
    RETURN -arg;
  ELSE
    RETURN arg;
  END IF;
END;
```

LOOP Statement

The LOOP statement repeats the execution of SQL statements. Since the LOOP statement itself has no condition to terminate the loop, a statement like LEAVE, RETURN, or SIGNAL is usually used to pass the flow of control outside of the loop.

Syntax

```
[ <loop_label>: ]
LOOP
  <statement>;
  [ ... ]
END LOOP [ <loop_label> ];
```

If the beginning label is specified, the label can be used with a LEAVE or ITERATE statement inside the LOOP statement. If the ending label is also specified, it needs to match the beginning label.

Example The following example repeats the execution 100 times, then exits from the loop using the LEAVE statement:

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;
  the_loop:
  LOOP
```

```
    ... -- do something here
    SET cnt = cnt + 1;
    IF cnt = 100 THEN
      LEAVE the_loop;
    END IF;
  END LOOP the_loop;
END;
```

REPEAT statement

The REPEAT statement repeats the statements until the specified condition returns true.

Syntax

```
[ <label>: ]
REPEAT
  <statement>;
  [ ... ]
UNTIL <condition>
END REPEAT [ <label> ];
```

In each iteration of execution, `<statement>`s are executed first, then `<condition>` is tested.

If the beginning label is specified, the label can be used with LEAVE or ITERATE statement inside the LOOP statement. If the ending label is also specified, it needs to match the beginning label.

Example The following example repeats the execution 100 times, then exits from the loop:

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;
  REPEAT
    ... -- do something here
    SET cnt = cnt + 1;
  UNTIL cnt = 100
  END REPEAT;
END;
```

WHILE Statement

The WHILE statement repeats the execution of SQL statements while the specified condition is true.

Syntax

```
[ <label>: ]
WHILE <condition> DO
<statement>;
[ ... ]
END WHILE [ <label> ];
```

In each iteration of execution, `<condition>` is first tested, and `<statement>`s are executed if `<condition>` is true.

If the beginning label is specified, the label can be used with a LEAVE or ITERATE statement inside the LOOP statement. If the ending label is also specified, it needs to match the beginning label.

Example The following example repeats the execution 100 times, then exits from the loop:

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;
  WHILE cnt < 100 DO
    ... -- do something here
    SET cnt = cnt + 1;
  END WHILE;
END;
```

FOR Statement

The FOR statement executes SQL statements for each object that qualified the specified SELECT query.

Syntax

```
[ <label>: ]
FOR <loop_variable> AS <select statement> DO
  <statement>;
  [ ... ]
END FOR [ <label> ]
```

<loop_variable> is used to qualify the names in the Select-list of <select statement> when they are used within the FOR body. And, <loop_variable> represents an object that is selected by <select statement>. You can access the selected object's attribute or invoke a method using <loop_variable>.

If the beginning label is specified, the label can be used with a LEAVE or ITERATE statement inside the LOOP statement. If the ending label is also specified, it needs to match the beginning label.

Example The following example counts the total length of the full name of all the artists with some threshold condition:

```
BEGIN
  DECLARE total INTEGER DEFAULT 0;
  DECLARE fname STRING;

  for_loop:
  FOR obj AS SELECT REF(a) FROM Artist a DO
    SET fname = obj.full_name();
    IF LENGTH(fname) > 20 THEN
      SET total = total + 20;
```

```
    ELSE
      SET total = total + LENGTH(fname);
    END IF;
  END FOR;
  RETURN total;
END;
```

The next example does the same thing using attribute access on the loop variable instead of method invocation `full_name()` above:

```
BEGIN
  DECLARE total INTEGER DEFAULT 0;
  DECLARE fname STRING;

  FOR obj AS SELECT REF(a) FROM Artist a DO
    SET fname = CONCAT(obj.firstName, obj.lastName);
    IF LENGTH(fname) > 20 THEN
      SET total = total + 20;
    ELSE
      SET total = total + LENGTH(fname);
    END IF;
  END FOR;
  RETURN total;
END;
```

The next example selects all the distinct ratings for each movie category, and returns it as a list:

```
BEGIN
  DECLARE ratings LIST(STRING) DEFAULT LIST(STRING)();

  FOR val AS SELECT DISTINCT category, rating FROM movie DO
    ADD (ratings, CONCAT(val.category, val.rating));
  END FOR;

  RETURN ratings;
END;
```

Note that these columns in the Select-list need to be qualified in the DO body using the loop variable `val`.

LEAVE Statement

The LEAVE statement passes the control flow out of a loop or a statement block.

Syntax

```
LEAVE label;
```

Use the label specified by FOR, LOOP, REPEAT, WHILE statement, or statement block

Example In the following example, the LEAVE statements moves the execution flow out of the outer loop directly from the inner loop:

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;
  outer_loop:
  WHILE cnt < 100 DO
    ... -- do something
    inner_loop:
    WHILE cnt < 200 DO
      ... -- do something
      SET cnt = cnt + 1;
      IF cnt >= 100 THEN
        LEAVE outer_loop; -- the control goes to line (A)
      END IF;
    END WHILE;
  END WHILE;
  ... -- line (A)
END;
```

ITERATE Statement

The ITERATE statement moves the execution flow back to the beginning of the loop and proceeds with the next iteration of the loop.

Syntax

```
ITERATE label;
```

Use the label specified by FOR, LOOP, REPEAT, or WHILE statement.

Example The following example uses the ITERATE statement to skip some cases in the iteration of the loop:

```
BEGIN
  DECLARE cnt, i INTEGER DEFAULT 0;
  SET i = 1;
  while_loop:
  WHILE cnt < 100 DO
    IF cnt = 50 THEN
      SET cnt = 90;
      ITERATE while_loop;
    END IF;
    ... -- do something with 'i'
    SET cnt = cnt + i;
  END WHILE;
END;
```

RETURN Statement

The RETURN statement returns the result of the method and exits from the method.

Syntax

```
RETURN [<expression> | NULL];
```

If the keyword RETURN is followed by nothing, it is equivalent to returning NULL.

If the RETURN statement is executed within a loop statement, e.g., WHILE or FOR, then the loop statement is terminated as well.

Example

The following statement block returns NULL if it finds an artist object without any biography:

```
BEGIN
  for_loop:
  FOR obj AS SELECT REF(a) FROM Artist a DO
    IF obj.biography IS NULL THEN
      RETURN NULL;
    END IF;
    ... -- do something else here
  END FOR;
END;
```

SET Assignment Statement

The assignment statement assigns a value to a variable.

Syntax

```
SET <variable> = <source expression> | NULL;
```

Type Compatibility

The data types of both `<source expression>` and the target `<variable>` need to be compatible. The data type compatibility for assignment is shown below. All the types listed in the same bullet are compatible with each other except list types.

- Numbers: BYTE, SHORT, INTEGER, LONG, FLOAT, DOUBLE, and NUMBER.
- STRING and TEXT
- CHARACTER
- TIMESTAMP
- DATE
- INTERVAL
- Multimedia types: AUDIO, IMAGE, VIDEO, and BYTES

- List type: A list type is compatible only with exactly the same type. For example, LIST(INTEGER) is compatible with LIST(INTEGER) but not compatible with LIST(LONG).
- Object: The target object type needs to be conformant with the source object, i.e., the class of the source object is the same or subclass of the target object's class.

Pass by Reference

When assigning a value of string, list type (e.g., LIST(INTEGER)), or multimedia types (e.g., BYTES or IMAGE), the assignment is done by passing its reference, not by copying its content.

Numeric Overflow

When assigning a number, an overflow exception could happen because of the lack of precision in the target type. For example, if you try to assign 1000000 to a variable of SHORT, Matisse will raise the numeric overflow exception.

SIGNAL Statement

The SIGNAL statement clears the diagnostic records and raises an exception, along with an optional text message. For more information about handling exceptions, see 13.7 Exception Handling.

Syntax

```
SIGNAL <exception_name> [SET MESSAGE_TEXT = <text
message>];
```

Example

See the example in the RESIGNAL statement below.

RESIGNAL Statement

The RESIGNAL statement resignals the exception along with an optional text message. It does not clear the diagnostic records, but raises the same exception again. The statement is used only within an exception handler.

Syntax

```
RESIGNAL;
```

Example

In the following example, it raises the `out_of_balance` exception, which will be caught by a handler. The handler will do some processing before reraising the same exception and exiting from the statement block.

```
BEGIN
  DECLARE out_of_balance CONDITION FOR CODE 2005;
  DECLARE EXIT HANDLER FOR out_of_balance
    SET ...;

  BEGIN  -- sub-block
    DECLARE CONTINUTE HANDLER FOR out_of_balance
      BEGIN
        ...        -- do something
```

```
        RESIGNAL;  -- reraise the same exception
      END;
    ...
    IF ... THEN
      SIGNAL out_of_balance;  -- raise an exception
    END IF;

  END;
END;
```

13.5 Selections in the Server

A selection is a collection of objects, which can be generated as the result of a SELECT statement execution or by reading successor objects of a relationship in an object in the server. For instance,

```
BEGIN
  DECLARE movies SELECTION(Movie);
  SELECT REF(m) FROM Movies m ... INTO movies;
  ... -- manipulation of 'movies'
END;
```

The next example copies `Starring` successors of a `Movie` object into a selection:

```
BEGIN
  DECLARE aMovie Movie;
  DECLARE actors SELECTION(Artist);

  SELECT REF(m) INTO aMovie FROM Movie m WHERE ...;
  SET actors = aMovie.Starring;
  ... -- manipulation of 'actors'
END;
```

Construct for Selections

There are several ways to construct selections that can be used in PSM. The following example shows two ways to assign an empty selection:

```
BEGIN
  DECLARE movies SELECTION(Movie) DEFAULT SELECTION();
  RETURN movies;
END;

BEGIN
  DECLARE actors SELECTION(Artist);
  SET actors = SELECTION();
  RETURN actors;
END;
```

The example below creates a selection using multiple selections

```
BEGIN
  DECLARE res SELECTION(movie);
  DECLARE someTitles SELECTION(movie);
  DECLARE moreTitles SELECTION(movie);
  SELECT REF(m) FROM movie m WHERE ... INTO someTitles;
  SELECT REF(m) FROM movie m WHERE ... INTO moreTitles;
  SET res = SELECTION(someTitles UNION moreTitles);
  -- another way
  SET res = SELECTION(someTitles, moreTitles);
  -- a more complex one
  SET res = SELECTION((someTitles UNION moreTitles)
                      INTERSECT otherTitles);
  -- another one
  SET res = SELECTION(someTitles EXCEPT moreTitles);
  RETURN res;
END;
```

This example creates a selection using multiple objects:

```
BEGIN
  DECLARE res SELECTION(movie);
  DECLARE mObj1, mObj2 movie;
  SELECT REF(m) INTO mObj1 FROM movie m WHERE ...;
  SELECT REF(m) INTO mObj2 FROM movie m WHERE ...;
  SET res = SELECTION(mObj1, mObj2);
  RETURN res;
END;
```

Methods for Selections

There are several system-defined methods for selections that can be used in PSM.

- ADD
- ADD_ALL
- CLEAR
- CONTAINS
- COUNT
- GET
- INSERT
- REMOVE
- REMOVE_AT

- REMOVE_DUPLICATES
- SET

ADD

Syntax `ADD(`*`object`*`)`

Purpose Add an object to the end of the selection.

Arguments *`object`*

The object to be added. If *`object`* is NULL, MATISSE_NULL_OBJECT error is returned.

Example

```
CREATE METHOD AddStarring(anArtist Artist)
RETURNS NULL
FOR Movie
BEGIN
  DECLARE s1 SELECTION(Artist);
  SET s1 = SELF.Starring;
  s1.ADD(anArtist);
  UPDATE SELF SET Starring = s1;
END;
```

ADD_ALL

Syntax `ADD_ALL(`*`selection`*`)`

Purpose Add objects in *`selection`* to the end of the selection.

Arguments *`selection`*

The objects to be added. If *`selection`* is NULL, MATISSE_NULL_OBJECT error is returned.

Example

```
CREATE METHOD AddStarring(artists SELECTION(Artist))
RETURNS NULL
FOR Movie
BEGIN
  DECLARE s1 SELECTION(Artist);
  SET s1 = SELF.Starring;
  s1.ADD_ALL(artists);
  UPDATE SELF SET Starring = s1;
END;
```

CLEAR

Syntax `CLEAR()`

Purpose Remove all of the elements in the selection.

CONTAINS

Syntax `CONTAINS(`*`object`*`)`

Purpose Determines whether *object* is contained in the selection

Arguments *object*

The object to locate in the selection. If *object* is NULL, the method returns false.

Example

```
CREATE METHOD HasActor(anArtist Artist)
RETURN BOOLEAN
FOR Movie
BEGIN
  DECLARE strr SELECTION(Artist);
  SET strr = SELF.Starring;
  RETURN strr.CONTAINS(anArtist);
END;
```

COUNT

Syntax `COUNT()`

Purpose Return the number of objects in the selection.

Example

```
BEGIN
  DECLARE movies SELECTION(Movie);
  DECLARE cnt INTEGER;
  SELECT REF(m) FROM Movies m WHERE ... INTO movies;
  SET cnt = movies.COUNT();
```

GET

Syntax `GET(`*`index`*`)`

Purpose Return the object at the specified position in this selection.

Arguments *index*

Index of object to be returned. If *index* is NULL, the method returns NULL. If index is out of range, the method returns MATISSE_ARG_OUTOFBOUNDS error. The index of the first object in a selection is 1.

Example

```
BEGIN
  DECLARE movies SELECTION(Movie);
  DECLARE aMovie Movie;
  DECLARE i INTEGER;

  SELECT REF(m) FROM Movie WHERE ... INTO movies;
  SET i = 1;
  WHILE i <= movies.COUNT() DO
    SET aMovie = movies.GET(i)
    ...
    SET i = i + 1;
  END WHILE;
END;
```

INSERT

Syntax `INSERT(index, object)`

Purpose Insert *object* at the specified position *index* in this selection.

Arguments *index*

Index at which the specified object is to be inserted. If index is out of range, the method returns MATISSE_ARG_OUTOFBOUND error. The index of the first object in a selection is 1.

object

Object to be inserted. If *object* is NULL, the method returns MATISSE_NULL_OBJECT error.

Example

```
CREATE METHOD AddActor(i INTEGER, anArtist Artist)
RETURNS NULL
FOR Movie
BEGIN
  DECLARE strr SELECTION(Artist);
  SET strr = SELF.Starring;
  strr.INSERT(i, anArtist);
  ...
END;
```

REMOVE

Syntax

```
REMOVE(object)
```

Purpose

Remove the first occurrence of the specified object in this selection. The method returns true if the specified object is found. Otherwise, it returns false.

Arguments

object

Object to be removed. If *object* is NULL, the method returns false.

Example

```
CREATE METHOD RemoveActor (anArtist Artist)
RETURNS BOOLEAN
FOR MOvie
BEGIN
  DECLARE strr SELECTION(Artist);
  DECLARE found BOOLEAN;

  SET strr = SELF.Starring;
  SET found = strr.REMOVE(anArtist);
  ...
  RETURN found;
END;
```

REMOVE_AT

Syntax

```
REMOVE_AT(index)
```

Purpose

Remove the object at the specified position in this selection. The method returns false if the specified position is out of range. Otherwise, it returns true.

Arguments

index

The index of the object to be removed. If *index* is NULL, the method does not remove any object and its return value is undetermined. The index of the first object in a selection is 1.

Example

```
CREATE METHOD RemoveActorAt (i INTEGER)
RETURNS NULL
FOR MOvie
BEGIN
  DECLARE strr SELECTION(Artist);

  SET strr = SELF.Starring;
  strr.REMOVE_AT(i);
  ...
END;
```

REMOVE_DUPLICATES

Syntax

```
REMOVE_DUPLICATES()
```

Purpose
Remove the duplicate objects in this selection.

Example

```
BEGIN
  DECLARE strr SELECTION(Artist);

  strr.REMOVE_DIPLICATES();
  ...
END;
```

SET

Syntax

```
SET(index, object)
```

Purpose
Replace the object at the specified position in this selection with the specified object.

Arguments
index

The index of the object to replace. The index of the first object in a selection is 1.

object

Object to be stored at the specified position. If *object* is NULL, the method returns MATISSE_NULL_OBJECT error.

Example

```
BEGIN
  DECLARE aMovie Movie;
  DECLARE strr SELECTION(Artist);
  DECLARE newActor Artist;

  SELECT REF(m) INTO aMovie FROM Movie m WHERE ...;
  SET strr = aMovie.Starring;
  strr.SET(1, newActor);
END;
```

13.6 Statement Blocks

A statement block is a group of SQL statements between the keywords BEGIN and END. Within a statement block, you can declare SQL variables and exception handlers.

Syntax

```
[label:]
BEGIN
  [<variable declaration> | <handler declaration>] [...]
  <SQL statement> [...]
END [label];

<variable declaration> ::=
  DECLARE <variable name> [, ... ] <type>
    [DEFAULT <literal constant>]

See  Declaration of Handler for the definition of <handler
declaration>.
```

If `label` is specified, it can be used with the LEAVE statement to exit from the statement block. If the optional ending `label` is specified, it needs to match the beginning label.

Variable Declaration

`<variable declaration>` defines local variables with names, a type, and an optional default value.

All the variable names need to be unique within a statement block. When statement blocks are nested, the inner block can see the variables declared in the outer block. If a variable V1 has the same name with another one, say V2, in outer statement block, V2 cannot be seen within the inner statement block. For example, the next statement block returns 10:

```
BEGIN
  DECLARE foo INTEGER;
  SET foo = 10;
  BEGIN
    DECLARE foo INTEGER;
    SET foo = 20; -- updating 'foo' in this block
  END;
  RETURN foo; -- returns 10, not 20
END;
```

All the available types for declaration are listed in *CREATE*.

All the variables are NULL until they are explicitly assigned a value, unless they are declared with DEFAULT clause.

Direct Execution of Statement Block

A statement block can be directly executed from the client application or within the `mt_sql` utility. The next example is executed in the `mt_sql` utility:

```
C:\>mt_sql -d exampledb@your_host
sql> BEGIN
  2>   DECLARE total NUMERIC(19, 2) DEFAULT 0.0;
  3>
  4>   loop_label:
  5>   FOR obj AS SELECT REF(e) FROM Employee e
  6>             WHERE location = 'SF'
```

```
  7>   DO
  8>     IF obj.expenses > 1000.0 THEN
  9>       -- max amount for each employee is 1000
 10>       SET total = total + 1000;
 11>     ELSE
 12>       SET total = total + obj.expenses;
 13>     END IF;
 14>   END FOR;
 15>
 16>   RETURN total;
 17> END;
10345.05
```

Returning Objects from Statement Block

A statement block can return a list of objects selected by a SELECT statement.

```
BEGIN
  DECLARE avg_len DOUBLE;
  DECLARE long_movies SELECTION(Movie);

  SELECT AVG(runningTime) INTO avg_len FROM Movie;
  SELECT REF(m) FROM Movie m
   WHERE runningTime > avg_len
    INTO long_movies;
    -- get the selected objects into a selection

  RETURN long_movies;
END;
```

If the example is executed in the mt_sql utility, the returned objects are saved in a selection named ‘DefaultSelection’, so you can do:

```
sql> SELECT * FROM DefaultSelection;
```

Returning a Table from Statement Block

A statement block returns a table result from a SELECT statement if the last instruction in the block is a SELECT statement.

```
BEGIN
  SELECT
    p.FirstName,
    p.LastName,
    p.IsInChargeOf.StartingYear,
    p.IsInChargeOf.EndingYear
  FROM Person p
  WHERE
    p.IsInChargeOf.EndingYear >= startYear
    AND p.IsInChargeOf.StartingYear <= endYear;
END;
```

13.7 Exception Handling

An exception handler specifies a set of statements to be executed when an exception occurs in a method or a statement block.

Declaration of Handler

To declare an exception handler, use the following form:

```
<handler declaration> ::=
  DECLARE <handler type> HANDLER FOR <exception conditions>
    <SQL statement>

<handler type> ::= CONTINUE | EXIT
```

Here is an example of `CONTINUE` handler, which sets a variable to -1 when the division-by-zero exception happens:

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR DIVISION_BY_ZERO
    SET cnt = -1;

  FOR obj AS SELECT REF(e) FROM Employee e DO
   -- division-by-zero exception may happen in the next line
    IF (obj.salary/obj.workHour) > 200 THEN
      SET cnt = cnt + 1;
    END IF;
  END FOR;

  RETURN cnt;
END;
```

Note that more than one declaration cannot have the same exception condition. For example, the following declarations are invalid:

```
-- sample of wrong code
BEGIN
  DECLARE EXIT HANDLER FOR MTEXCEPTION
    SET res = 0;
  DECLARE EXIT HANDLER FOR MTEXCEPTION
    SET another = 10;
  ...
END;
```

Each handler can contain up to 16 exception conditions.

Handler Types

Matisse supports two types of handlers: `CONTINUE` and `EXIT`.

- EXIT: After the handler is executed successfully, the control is returned to the end of the statement block that declared the handler.

- CONTINUE: After the handler is executed successfully, the control is returned to the SQL statement that follows the statement that raised the exception. Note: If the statement that raised the exception is in a FOR, IF, WHILE, LOOP, or REPEAT statement, the control goes to the statement that follows END FOR, END IF, END WHILE, END LOOP, or END REPEAT, unless the handler is defined inside these loop statements.

In the following example, if the division-by-zero error happens at line (A), then the exception handler is executed and the control goes to line (B), i.e., exits from the FOR loop.

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR DIVISION_BY_ZERO
    SET cnt = -1;

  FOR obj AS SELECT REF(e) FROM Employee e DO
    IF (obj.salary/obj.workHour) > 200 THEN -- line (A)
      SET cnt = cnt + 1;
    END IF;
  END FOR;

  RETURN cnt; -- line (B)
END;
```

The following example declares the exception handler within the FOR loop. If the division-by-zero error happens at line (A), then the exception handler is executed and the control goes to line (B), i.e., does not exit from the FOR loop.

```
BEGIN
  DECLARE cnt INTEGER DEFAULT 0;

  FOR obj AS SELECT REF(e) FROM Employee e DO
    DECLARE CONTINUE HANDLER FOR DIVISION_BY_ZERO
      SET cnt = -1;

    IF (obj.salary/obj.workHour) > 200 THEN -- line (A)
      SET cnt = cnt + 1;
    END IF;
    ... -- line (B)
  END FOR;

  RETURN cnt;
END;
```

User Defined Exceptions

You can define an user exception in a method or a statement block, which can be used to raise an exception using the SIGNAL statement. The form to declare a user defined exception is:

```
DECLARE <exception-name> CONDITION
  [FOR <user-exception-code>];
```

If `<user-exception-code>` is not provided, the code is set to 0.

Here is an example, which declares a user defined exception and defines a handler for it as well:

```
DECLARE too_many_elements CONDITION FOR CODE 1002;
DECLARE CONTINUE HANDLER FOR too_many_elements
   ...;
```

An exception name needs to be unique within a statement block.

Unhandled Exception

If an exception is not handled by anyone, the unhandled exception is returned to the client application that called the method or the statement block.

For example, if a method raised DIVISION_BY_ZERO exception and is not handled by anyone, then the client API that called the method, e.g., executeQuery() for Java or MtSQLExecDirect() for C, returns the MATISSE_DIVISION_BY_ZERO exception.

If an user defined exception is not handled, then the client returns the MATISSE_USER_EXCEPTION error. In order to get more information about the user exception, use the C API MtSQLGetParamValue() or equivalent in other language bindings. The second parameter for MtSQLGetParamValue() can be one of the followings:

- MTSQL_USER_EXCEPTION_NAME, to get the name of the user exception
- MTSQL_USER_EXCEPTION_CODE, to get the code of the user exception
- MTSQL_USER_EXCEPTION_MESSAGE, to get the text message of the user exception. If no text message was specified by the SIGNAL statement, then you get MT_NULL as its return type, not MT_STRING.

13.8 Using Lists

You can use list types, e.g., list of integer, in SQL methods or statement blocks. An element in a list is accessible using square brackets ([]), and elements can be added/removed from a list using ADD, REMOVE, or INSERT functions. You can also use the list functions AVG, MIN, MAX, SUM, COUNT, ELEMENT and SUBLIST presented in *section 10.3, List Functions*.

Access using brackets

You can access an element of a list at a specific place using square brackets list many other programming languages. The following example returns the second element in a list:

```
BEGIN
```

```
    DECLARE aList LIST(INTEGER)
      DEFAULT LIST(INTEGER)(10, 20, 30);
    RETURN aList[2]; -- will return '20'
  END;
```

If the subscript is out of bounds, an error is returned.

Set list assignment

You can assign a list to a list variable with the SET function. The following example returns an empty list:

```
BEGIN
  DECLARE aList LIST(INTEGER); -- no value
  SET aList = LIST(INTEGER)(10, 20, 30);
  SET aList = LIST(INTEGER)();
  RETURN aList; -- will return 'an empty list'
END;
```

You can also use the brackets expression combined with the SET function for replacing an element at a specific location. The following example replace the third element in the list:

```
BEGIN
  DECLARE aList LIST(INTEGER);
  SET aList = LIST(INTEGER)(10, 20, 30);
  SET aList[3] = 35;
  RETURN aList[3]; -- will return '35'
END;
```

If the subscript is out of bounds, an error is returned.

ADD

To add an element to a list, use the ADD function.

```
ADD(list, element|list)
```

The function adds the new element to the end of the list. If the new element to be added is NULL, the function returns the MATISSE_NULL_OBJECT error. The following example adds an element to a list, a list to a list and then returns the updated list:

```
BEGIN
  DECLARE aList LIST(INTEGER) DEFAULT LIST(INTEGER)(1, 2);
  DECLARE bList LIST(INTEGER) DEFAULT LIST(INTEGER)();
  ADD(bList, 3);
  ADD(aList, bList);
  RETURN aList; -- will return '(1,2,3)'
END;
```

INSERT

To insert an element to a list, use the INSERT function.

```
INSERT(list, element, n)
```

The function inserts the new element before the *n*-th element in the list. If *n* is less than 1or more than the number of elements in the list, it raises an out-of-bounds error. If the new element to be inserted is NULL, the function returns the MATISSE_NULL_OBJECT error. The following example insert an element into a list and then returns the updated list:

```
BEGIN
  DECLARE aList LIST(INTEGER) DEFAULT LIST(INTEGER)(3);
  INSERT(aList, 1, 1);
  INSERT(aList, 2, 2);
  RETURN aList; -- will return '(1,2,3)'
END;
```

REMOVE

To remove an element in a list, use the REMOVE function.

```
REMOVE(list, n)
```

The function removes an element at the *n*-th position in the list. If *n* is less than 1or more than the number of elements in the list, it raises an out-of-bounds error. The following example remove an element from a list and then returns the updated list:

```
BEGIN
  DECLARE aList LIST(INTEGER)
    DEFAULT LIST(INTEGER)(1, 2, 3);
  REMOVE(aList, 3);
  RETURN aList; -- will return '(1,2)'
END;
```

Using other list functions

The following example shows the AVG, MIN, MAX, SUM, COUNT, ELEMENT and SUBLIST functions used in block statement:

```
BEGIN
  DECLARE aList LIST(INTEGER);
  DECLARE res INTEGER;
  SET aList = LIST(INTEGER)(1,2,3);
  SET res = AVG(aList);
  SET res = MIN(aList);
  SET res = MAX(aList);
  SET res = SUM(aList);
  SET res = COUNT(aList);
  SET res = ELEMENT(aList, 2); -- equivalent to aList[2]
  RETURN SUBLIST(aList, 2, 1);
END;
```

13.9 System Defined Methods

Currently, Matisse SQL provides two system defined methods.

- isMetaSchema() is defined for the class MtClass. It returns true if the class is part of the meta schema, for example the class MtAttribute. Otherwise, it returns false.
- isPredefinedObject() is defined for the class MtObject. It returns true if the object is one of the objects that are generated when the database is initialized. Otherwise, it returns false.

For example, you can select all the user defined classes with the following SQL statement:

```
SELECT * FROM MtClass c WHERE c.isMetaSchema() = false;
```

13.10 Debugging Methods

The PSM_OUTPUT module allows developers to easily trace the execution of stored methods. The functions in this module enable you to print out variable name, type and content as well as messages from SQL methods into the database log file. The PRINT function prints out an expression value. The PRINT_LINE function prints out an expression value and then terminates the line. The PRINT_VARIABLE function prints out detailed information about a variable (name, type and value) and then terminate the line. The ENABLE function enables calls to PRINT, PRINT_LINE and PRINT_VARIABLE. Calls to these functions are ignored if the PSM_OUTPUT module is not enabled. The DISABLE function disables calls to PRINT, PRINT_LINE and PRINT_VARIABLE, and purges the message buffer of any remaining information.

PSM_OUTPUT

Syntax

```
PSM_OUTPUT.PRINT(<expression>)
PSM_OUTPUT.PRINT_LINE(<expression>)
PSM_OUTPUT.PRINT_VARIABLE(<expression>)
PSM_OUTPUT.ENABLE()
PSM_OUTPUT.DISABLE()
```

Example

```
BEGIN
  DECLARE Obj Person;
```

```
  [...]
  PSM_OUTPUT.PRINT(Obj.LastName);
  PSM_OUTPUT.PRINT_LINE('');
  PSM_OUTPUT.PRINT_LINE(CONCAT('printTrace() - ',
Obj.FirstName));
  PSM_OUTPUT.PRINT_VARIABLE(obj.OID);
END
```

An excerpt of the log file after running the SQL statement above:

24 Oct. 2007 18:03:58 PSM Output: Washington
24 Oct. 2007 18:03:58 PSM Output: printTrace() - Georges
24 Oct. 2007 18:03:58 PSM Output: V2 (MT_OID) = 0x1086

The following example disables calls to PRINT, PRINT_LINE and PRINT_VARIABLE, and purges the message buffer of any remaining information:

```
BEGIN
  PSM_OUTPUT.DISABLE();
END
```

The example below enables calls to PRINT, PRINT_LINE and PRINT_VARIABLE:

```
BEGIN
  PSM_OUTPUT.ENABLE();
END
```

14 Options

14.1 Setting Options

MEMORY_QUOTA

SET MEMORY_QUOTA statement can be used to set the maximum memory that an SQL execution can use in the server. For example,

```
SET MEMORY_QUOTA 100M;
```

The above statement sets the memory quota to 100 Megabyte.

A memory quota is effective per connection to the server. Once memory quota is set to some value, the memory quota is effective until the connection to the server is closed.

The default memory quota is 500 MB. The minimum memory quota is 50 MB.

CONNECTION_OPTION

Connection options affect the way you can interact with the database. These options allows you to specify the type of access to the database, the object locking policy as well as the amount of time the server waits for access conflicts to be resolved.

Syntax

```
SET CONNECTION_OPTION DATA_ACCESS_MODE { DEFAULT
                                        | DATA_READONLY
                                        | DATA_MODIFICATION
                                        | DATA_DEFINITION
                                       }
```

Example

```
SET CONNECTION_OPTION DATA_ACCESS_MODE DATA_DEFINITION;
UPDATE schema objects ...;
COMMIT;
```

This option allows you to specify the type of access that you intend to use when connecting to the database. Possible values are:

- DATA_READONLY allows read only access to the data objects and to the schema. Any attempt to start a transaction will fail (only SET TRANSACTION READ ONLY is allowed).
- DATA_MODIFICATION (default) allows read/write access to the data objects and read only access to the schema.

- DATA_DEFINITION allows read/write access to the data objects and to the schema.

The first two access modes optimize the access to the schema. The DATA_DEFINITION access mode must be used only when schema or meta-schema updates are necessary.

This option cannot be changed when the transaction is in progress.

Syntax

```
SET CONNECTION_OPTION LOCKING_POLICY { DEFAULT
                                      | DEFAULT_ACCESS
                                      | ACCESS_FOR_UPDATE
                                   }
```

Example

```
SET CONNECTION_OPTION LOCKING_POLICY ACCESS_FOR_UPDATE;
```

This option allows the server to be configured to handle requests for read locks using write locks instead. The possible values are:

- DEFAULT_ACCESS (default): Normal behavior, requests for read locks result in read locks.
- ACCESS_FOR_UPDATE: Requests for read locks result in write locks.

This option may be changed at any time.

Syntax

```
SET CONNECTION_OPTION LOCK_WAIT_TIME { DEFAULT
                                      | NO_WAIT
                                      | WAIT_FOREVER
                                      | number
                                    }
```

Example

```
SET CONNECTION_OPTION LOCK_WAIT_TIME 500;
```

This option allows you to specify the amount of time the server waits for access conflicts to be resolved; if a time-out occurs (wait-time expires), the explicit or implicit lock request is rejected. The possible values are:

- NO_WAIT: If the lock cannot immediately be granted, the lock request is released and the function returns immediately.
- WAIT_FOREVER (default): The server waits until there is a deadlock or until the lock is granted.
- A positive integer: This is the time (in milliseconds) that the server waits for the lock to be granted. If the wait-time expires, the lock request is rejected. If a deadlock occurs, the transaction fails or the lock request is rejected.

When multiple objects are requested, the wait-time applies to each object request individually. The wait-time affects the process of obtaining locks for reads and writes within transactions. Object version requests are affected neither by locks nor by wait-times.

Appendix A

Appendix A Sample Application Schema

This appendix describes the sample application schema most commonly used throughout the SQL examples in the previous sections. The schema is described in the Matisse ODL format.

```
interface Movie : persistent
{
        attribute String Name;
        mt_entry_point_dictionary "MovieNameDict"
          entry_point_of Name
          make_entry_function "make-entry";
        attribute String Title;
        mt_entry_point_dictionary "MovieTitleDict"
          entry_point_of Title
          make_entry_function "make-full-text-entry";
        attribute String Synopsis;
        mt_entry_point_dictionary "MovieSynopsisDict"
          entry_point_of Synopsis
          make_entry_function "make-full-text-entry";
        attribute String Rating;
        attribute String Category;
        attribute Long RunningTime = Long(0);
        attribute Long rankForWeek;
        attribute Image Thumbnail;
        attribute Video Preview;

        relationship Set<MovieDirector>
                directedBy [0, 1]
                inverse MovieDirector::Direct;

        relationship List<Artist>
                Starring [0, -1]
                inverse Artist::Biography;

        relationship Set<boxOffice>
                boxOfficeRecords [0,-1]
                inverse boxOffice::topTitles;
};

interface Artist : persistent
```

```
{
   attribute String LastName;
   attribute String FirstName;
   mt_entry_point_dictionary "LastNameDict"
          entry_point_of LastName
          make_entry_function "make-entry";

   relationship Set<Movie>
                Biography [0, -1]
                inverse Movie::Starring;
};

interface MovieDirector : Artist : persistent
{
   relationship Set<Movie> Direct
                inverse Movie::directedBy;
};

interface boxOffice : persistent
{
        attribute Date week;
        attribute Long totalReceipts = Long(0);
        attribute List<Numeric(10, 2)> topReceipts;
        relationship Set <Movie>
                topTitles [0,50]
                inverse Movie::boxOfficeRecords;
};

interface Theater : persistent
{
  attribute String Name;
  relationship List<Movie> playingMovies;
};
```

Appendix B Using Matisse SQL with C-APIs

This section describes how to use Matisse SQL with Matisse C-APIs.

A SQL statement manipulates object instances of Matisse classes, which are qualified by their class name.

A SQL statement can access both the relationships and the attributes of Matisse objects. The attributes and relationships of the object instances of a class are the attributes and relationships defined for the class itself as well as the attributes and relationships defined on all the superclasses from which the class inherits.

The execution of a SQL `SELECT` statement produces a projection for the columns defined in the Select-list.

Here is a simple example using the C API:

```
MtSTS sts;
MtSQLStmt stmt;
MtStream stream;

/* initialization */
sts = MtSQLAllocStmt (&stmt);

/* execute a SQL statement */
sts = MtSQLExecDirect (stmt, "SELECT * FROM person");
if ( MtFailure(sts) )
 printf ("Error!! code = %d, message = %s\n", sts,
        MtError());

/* open a row stream on statement */
sts = MtSQLOpenStream (&stream, stmt);

/* get the row value for the first column */
MtSQLNext (stream);
MtSQLGetRowValue(stream, 1, ...);

/* clean up */
sts = MtCloseStream (stream);
sts = MtSQLFreeStmt (stmt);
```

Please refer to the *Matisse C API Reference* for more detailed information on how to use Matisse SQL C APIs.

Index

Symbols

A

B

C

G

H

I

J

L

M

N

S

T

U

V

W