Matisse® SQL Programmer's Guide

14th Edition

May 2003

Matissse SQL Programmer's Guide

Matisse Software Inc.
433 Airport Blvd, Suite 421
Burlingame, CA 94010
USA

Printed in USA.

PDF generated 3 May 2003

Contents

Introduction 9

Conventions 9

1 Data Accessed with Matisse SQL 10

2 The mt_sql Utility 11

2.1 Simple Example 11

2.2 Basic Usage 11

2.3 Command Line Options 12

2.4 Online Help 13

2.5 Discovering the Schema 13

3 Constants and Identifiers 15

3.1 What Is a Constant? 15

Integer Constants 15

Numeric Constants 15

Real Constants 16

Boolean Constants 16

Character String Constants 16

Date and Timestamp Constants 17

Time Interval Constants 18

Bytes Constants 18

List Constants 18

3.2 What Is an Identifier? 19

3.3 What Is a Keyword? 19

4 Selecting Data 22

4.1 Using the SELECT Command 22

Using the ONLY Keyword 22

Specifying a SQL Projection 23

OID, REF, and Relationship in SQL Projection 24

CLASS_NAME and CLASS_ID 25

4.2 Join Operation 26

Natural Join 26

Conditional Join 27

Sorting the Result 27

4.3 Creating a SQL Selection Using the INTO Keyword 27

4.4 Deleting a Selection Result 28

4.5 Specifying a Search Criteria with WHERE 29

4.6 Using Attributes in Expressions 29

Specifying an Attribute in a WHERE Clause 29

4.7 Combining Predicates with AND and OR 30

Precedence of Evaluation of AND and OR 31
4.8 Specifying a Negative Condition with NOT 32
4.9 Specifying a Type Predicate with IS OF 32
4.10 Getting DISTINCT Values 33
4.11 Specifying Sort Criteria with ORDER BY 33
4.12 Grouping with GROUP BY and HAVING 34
Grouping by class 35
HAVING clause 35

5 Using Numeric Values 37
5.1 Introduction 37
5.2 Comparison Operators 37
5.3 Performing Arithmetic Operations 37
Expressions and Arithmetic Operators 37
Evaluating an Expression: An Example 38
5.4 Result Types from Arithmetic Expressions 38
5.5 Performing an Interval Test 40
5.6 Using the ANY and ALL Keywords 40

6 Using Null Values 41
6.1 Introduction 41
6.2 What Is a Null Value? 41
6.3 The IS NULL Keyword 41
Example: Comparison with Null Values 41

7 Using Text Values 43
7.1 Introduction 43
7.2 What Does Text Comparison Mean? 43
7.3 What Is a Pattern? 45
7.4 How to Use the % Wildcard Character 45
7.5 How to Use the Underscore Wildcard Character 46
7.6 Specifying a Pattern with the LIKE Keyword 46
7.7 How to Use an Escape Character 47
7.8 Using the ANY and ALL Keywords 47
Quantified Comparison with the ANY Keyword 48
Comparison with the ALL Keyword 48
Equivalent Comparisons 48
Alternate Syntax 48
Examples 49
7.9 Selecting Objects by Entry Points 49
Exact Match Search 49
Pattern Matching 50

8 Using Relationships 51
8.1 Introduction 51
8.2 What Is a Relationship? 51

8.3 The IN Keyword 51
Comparing with a List of Successors 52
8.4 Navigational Queries 52
Using a Single Relationship in the SELECT List 52
Using Relationships and Other Columns in the SELECT List 53
Using a Relationship in the WHERE Clause 53
Relationship COUNT 54
Dealing with Empty Relationships 54

9 Version Travel 56
9.1 Introduction 56
9.2 Specifying a Version Travel Query 56

10 Managing Transactions and Versions 58
10.1 Introduction 58
10.2 Starting a Version Access 58
10.3 Ending a Version Access 59
10.4 Starting a Transaction 59
10.5 Committing a Transaction 59
10.6 Cancelling a Transaction 60

11 SQL Functions 61
11.1 Character String Functions 61
CONCAT 61
INSTR 62
LENGTH 63
LOWER 63
LTRIM 63
RTRIM 64
SUBSTR 65
UPPER 65
11.2 List Functions 66
AVG 66
ELEMENT 67
MAX 67
MIN 68
SUBLIST 68
SUM 69
COUNT 69
LIST 70
11.3 Set Functions 70
AVG 70
COUNT 71
MAX 71
MIN 72
SUM 72

11.4 Set functions for relationship aggregation 73
AVG 73
COUNT 74
MAX 74
MIN 74
SUM 75
11.5 Datetime Functions 75
CURRENT_DATE 75
CURRENT_TIMESTAMP 75
EXTRACT 76
11.6 Conversion Functions 76
CAST 76

12 Defining a Schema 79
12.1 Classes, Attributes, and Relationships 79
CREATE 79
ALTER 85
DROP 86
12.2 Indexes 87
CREATE 87
DROP 87
12.3 Entry Point Dictionaries 88
CREATE 88
DROP 88
12.4 Methods 89
CREATE 89
DROP 90
COMPILE 90

13 Manipulating Data 92
13.1 Updating Data 92
UPDATE 92
13.2 Inserting Data 95
INSERT 95
13.3 Deleting Data 96
DELETE 96

14 Stored Methods and Statement Blocks 97
14.1 A Simple Example 97
14.2 Method Invocation 98
Calling a Method in SELECT Statement 98
Calling a Method in Method Body 98
Calling a Static Method 99
Static Method and Query Optimization 99
14.3 Control Statements 100
IF Statement 100

LOOP Statement 101
REPEAT statement 102
WHILE Statement 102
FOR Statement 103
LEAVE Statement 104
ITERATE Statement 105
RETURN Statement 105
SET Assignment Statement 106
SIGNAL Statement 107
RESIGNAL Statement 107
14.4 Statement Blocks 107
Variable Declaration 108
Direct Execution of Statement Block 108
Returning Objects from Statement Block 109
14.5 Exception Handling 109
Declaration of Handler 109
Handler Types 110
User Defined Exceptions 111
Unhandled Exception 111
15 Options 113
15.1 Setting Options 113
MAXOBJECTS 113
Appendix A Sample Application Schema 114
Index 116

Tables

Table 2.1	Command Line Options	12
Table 3.1	Keywords for Formulating SQL Requests	19
Table 4.1	Comparison of OID and REF()	25
Table 4.2	AND Operator Truth Table	31
Table 4.3	OR Operator Truth Table	31
Table 4.4	Equivalent Logical Expressions	31
Table 5.1	Comparison Operators	37
Table 5.2	Types Resulting from Arithmetic Operation	39
Table 5.3	Type Resulting from the Negation Operation	39
Table 6.1	IS [NOT] NULL	42
Table 7.1	Text Comparison Operators	43
Table 7.2	ASCII Characters and Their Numeric Values	44
Table 7.3	Equivalent Expressions Using ANY and ALL	48
Table 11.1	Supported casts between built-in data types	77

Introduction

This manual describes the syntax and usage of the Matisse SQL language. Matisse SQL allows you to select a subset of the instances of a class that meet certain criteria regarding the attribute values or the relationships.

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 Data Accessed with Matisse SQL

This section describes Matisse SQL with respect to the Matisse development environment.

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 SQL C APIs.

2 The mt_sql Utility

The `mt_sql` utility is Matisse's command-line interface allowing you to interactively execute SQL statements and display the result.

2.1 Simple Example

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.

2.2 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 a 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
```

2.3 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 2.1*.

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

Table 2.1 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;
```

2.4 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.
    ...
```

2.5 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
```

3 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

3.1 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.

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 maximum length of a character string constant is 2000 characters. Character strings are case sensitive.

The following example shows how to enter a character string containing a single quote:

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

Note that this string has the value Computer's and has a length of 10.

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, 1997, you could write the following predicate:

```
birthdate > DATE '1997-10-01'
```

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, 1997 at 11:30 AM with the following predicate:

```
lastEntry >
   TIMESTAMP '1997-10-01 11:30:00'
```

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

```
lastEntry >
   TIMESTAMP '1997-10-01 11:30:00.00'
lastEntry >
   TIMESTAMP '1997-10-01 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 '1997-10-01 11:30:00'
TIMESTAMP '1997-10-01 11:30:00' AT LOCAL
TIMESTAMP '1997-10-01 20:30:00' AT GMT
TIMESTAMP '1997-10-01 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, 1997-10-01 20:30:00, in UTC, the following predicates would evaluate to true:

```
lastEntry =
   TIMESTAMP '1997-10-01 11:30:00' AT LOCAL
lastEntry =
   TIMESTAMP '1997-10-01 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.

3.2 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.

3.3 What Is a Keyword?

A keyword is defined by the Matisse SQL language. *Table 3.1* lists the keywords that you can use to formulate an SQL request. Matisse SQL keywords are not case sensitive.

Table 3.1 Keywords for Formulating SQL Requests

<	>	*	/
‘	“	.	,
;	(	)	@

Table 3.1 Keywords for Formulating SQL Requests *(Continued)*

+	-	=	ADD
ALL	ALTER	AND	ANY
AS	ASC	AT	ATTRIBUTE
AUDIO	AVG	BEGIN	BETWEEN
BIGINT	BLOB	BOOLEAN	BOOLEAN LIST
BY	BYTE	BYTE ARRAY	BYTES
CALL	CARDINALITY	CASE	CAST
CHAR	CHARACTER	CHAR_LENGTH	CLASS
CLASS_ID	CLASS_NAME	CLOB	COMMIT
COMPARE	COMPILE	COMPILED	CONCAT
CONNECT	CONNECTION	COUNT	CREATE
CURRENT	CURRENT_DATE	CURRENT_TIMESTAMP	DATE
DATE LIST	DECIMAL	DECLARE	DEFAULT
DELETE	DELETED	DESC	DICTIONARY
DISCONNECT	DO	DOUBLE	DOUBLE ARRAY
DOUBLE LIST	DROP	DUPLICATE	ELEMENT
ELSE	ELSEIF	END	ENTRY_POINT
ENUM	ENUM LIST	ESCAPE	EVENT
EXCEPT	FALSE	FLOAT	FLOAT ARRAY
FLOAT LIST	FOR	FROM	GMT
GROUP	HAVING	IF	IMAGE
IN	INDEX	INHERIT	INNER
INOUT	INSERT	INSERTED	INSTANCE
INSTR	INTEGER	INTEGER ARRAY	INTEGER LIST
INTERSECT	INTERVAL	INTERVAL LIST	INTO
INVERSE	IS	JOIN	LEAVE
LENGTH	LIKE	LIST	LOCAL
LONG	LONG LIST	LOWER	LTRIM
MAKE_ENTRY	MAX	MAXOBJECTS	METHOD
METHODS	MIN	NATURAL	NOT
NOTIFY	NULL	NUMERIC	NUMERIC LIST

Table 3.1 Keywords for Formulating SQL Requests *(Continued)*

OF	OFF	OID	ON
ONLY	OR	ORDER	OUT
PRECISION	READ	READONLY	REAL
REF	REFERENCES	RELATIONSHIP	RETURN
RETURNS	ROLLBACK	RTRIM	SELECT
SELECTION	SELF	SENSITIVE	SET
SHORT	SHORT ARRAY	SHORT LIST	SMALLINT
STATIC	STRING	STRING ARRAY	STRING LIST
SUBLIST	SUBSCRIBE	SUBSTR	SUBSTRING
TIMESTAMP	TIMESTAMP LIST	UNKNOWN	UNSUBSCRIBE
UPDATE	UPDATED	UPPER	USER
UTC	VALUES	VARCHAR	VARYING
VERSION	VIDEO	WAIT	WHERE
WORK	WRITE		

4 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

4.1 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 [DISTINCT] { *
    | [<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 }
```

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 "proper objects" 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 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, m.directedBy.lastName FROM movie m;
SELECT AVG(runningTime) FROM movie;
```

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
```

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 keyword `AS` is optional and may be omitted.

OID, REF, 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`.

```
sql> 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.

```
sql> SELECT m.title, a.name AS "Starring Artists"
  2>   FROM movie m, artist a
  3>   WHERE m.starring = a.OID;
title           Starring Artists
--------------- ----------------
The Green Mile  Tom Hanks
Titanic         L. DiCaprio
Titanic         Kate Winslet
...
```

See the next section for the join operation.

`REF()` can be used in SQL projection. `REF()` exports objects as the C API type `MtOid`, and is used for passing objects from SQL to the language bindings. The following table compares OID and REF(). For example:

```
SELECT REF(movie) FROM movie;
```

```
SELECT REF(m.starring) FROM movie m;
```

Table 4.1 Comparison of OID and REF()

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

CLASS_NAME and CLASS_ID

Matisse SQL provides two other pseudo attributes besides OID, CLASS_NAME and CLASS_ID.

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

```
SELECT LastName, CLASS_NAME FROM Artist;

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

CLASS_NAME 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.CLASS_NAME LIKE '%Corporate%';
```

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

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

The following predicate executes faster:

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

For more information about the IS OF predicate, refer to *section 4.9, Specifying a Type Predicate with IS OF*.

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

```
SELECT LastName, CLASS_ID FROM Artist;

LastName             CLASS_ID
-------------------- --------------------
Hanks                0x25f0
Foster               0x25f0
```

```
Spielberg           0x260c
```

The type of CLASS_ID is string.

4.2 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 4.11, Specifying Sort Criteria with ORDER BY*) attributes of the first class specified in the `FROM` clause.

4.3 Creating a SQL Selection Using the INTO Keyword

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

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;
```

After executing this 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;
```

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

4.4 Deleting a Selection Result

.

SQL Selections created with the INTO keyword as shown in the previous 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;
```

4.5 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.

4.6 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;
```

4.7 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 4.2* is the truth table for the AND operator.

Table 4.2 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 4.3 is the truth table for the OR operator.

Table 4.3 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 4.4*.

Table 4.4 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);
```

4.8 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%';
```

4.9 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.

4.10 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 kinds of ratings for each category:

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

Current limitation: When you specify DISTINCT in a SELECT statement, you can select only scalar values without relationship navigation, i.e., you cannot select list types, multimedia types, or any relationship navigation.

4.11 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:

```
{ attribute [ASC | DESC] }
```

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
```

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

4.12 Grouping with GROUP BY and HAVING

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 attribute [, ...]
  [HAVING <search condition>]
```

A GROUP BY clause can have up to 16 attributes as its grouping criteria.

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 not valid:

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

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

Restrictions on grouping attribute

1. Grouping criteria cannot contain relationship navigation.
2. Grouping attribute cannot be of list types nor media types, e.g., LIST(INTEGER), IMAGE, or VIDEO

Grouping by class

CLASS_NAME or CLASS_ID can be used to group objects by their class. For example,

```
SELECT CLASS_NAME, AVG(salary) FROM Employee
 GROUP BY CLASS_NAME;

CLASS_NAME           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`, not including objects of class `Manager`, which is a subclass of `Employee`.

Note that CLASS_NAME and CLASS_ID have the identical effect for grouping.

HAVING clause

The HAVING clause restricts the groups of the selected objects to those groups for which the <search condition> is true.

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

```
SELECT category, AVG (runningTime) FROM Movie
 GROUP BY category
```

```
 HAVING AVG (runningTime) > 120;

category          avg
----------------- ------------
            Drama        125.1
```

The HAVING clause can reference only

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

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.

5 Using Numeric Values

5.1 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

5.2 Comparison Operators

- *Table 5.1* lists the various comparison operators that are available:

Table 5.1 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.

5.3 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.

5.4 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 5.2*.

Table 5.2 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 5.3*.

Table 5.3 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.

5.5 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;
```

5.6 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 7.8, Using the ANY and ALL Keywords*.

6 Using Null Values

6.1 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

6.2 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.

6.3 The IS NULL Keyword

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

The syntax for evaluation 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 6.1*.

Table 6.1 IS [NOT] NULL

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

7 Using Text Values

7.1 Introduction

Some topics covered in this section are similar to those presented in *section 5, 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

7.2 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 7.1*.

Table 7.1 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 7.2*.

Table 7.2 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 7.2 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);
```

7.3 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

7.4 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 ky.

7.5 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’.

7.6 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 '%-%';
```

7.7 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\.

When the escape character does not precede a special character, it is ignored. For example, if `\` is the escape character, the character string `A\BC` is the same as the character string `ABC`.

7.8 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 7.3* lists these equivalences.

Table 7.3 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'
```

7.9 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%';
```

8 Using Relationships

8.1 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

8.2 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.

8.3 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 nothing or *`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 commands, which are equivalent:

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

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

8.4 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;
```

9 Version Travel

9.1 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 10, Managing Transactions and Versions*.

9.2 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);
```

10 Managing Transactions and Versions

10.1 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

10.2 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 August1994;
```

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 `August1994` was saved at the logical time `2A` (in hexadecimal), the statement from the previous example could be expressed as follows:

```
SET TRANSACTION READ ONLY August19940000002A
```

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.

10.3 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.

10.4 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;
```

10.5 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 August1994;
```

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 `August1994`.

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.

10.6 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
```

11 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:

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

11.1 Character String Functions

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

- `CONCAT`
- `LOWER`
- `LTRIM`
- `RTRIM`
- `SUBSTR (SUBSTRING)`
- `UPPER`

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

- `INSTR`
- `LENGTH (CHAR_LENGTH)`

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
```

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
```

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
```

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
```

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
```

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
```

11.2 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 11.2, 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));
```

11.3 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}.]*)
COUNT ([{class | alias}.] relationship [.succ_class].* )
```

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 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;
```

11.4 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 11.2, 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.

11.5 Datetime Functions

This section explains the following datetime functions.

- CURRENT_DATE
- CURRENT_TIMESTAMP
- EXTRACT

CURRENT_DATE

Syntax `CURRENT_DATE`

Purpose Returns the current date in the Universal Coordinated Time zone.

CURRENT_TIMESTAMP

Syntax `CURRENT_TIMESTAMP`

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
```

11.6 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 11.1* 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 11.1 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!!
```

12 Defining a Schema

This section explains the SQL statements that are used to define a database schema, that is, those that define 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.

12.1 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} class
  [{UNDER | INHERIT} superclass [, ...]]
(
  <property> [, ...]
  <class_constraint> [, ...]
)

<property> ::=
    <attribute_name> <attribute_type>
       [DEFAULT <literal>] [NOT NULL] |
    <relationship_name> [READONLY]
      {RELATIONSHIP | REFERENCES} [LIST | SET]
        (<succ_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>)]| CLOB [(<max>)] |
       BOOLEAN |
       BYTE | TINYINT
       SHORT | SMALLINT
       INTEGER | INT
       LONG | BIGINT
       NUMERIC [(<precision>[,<scale>])]|
       FLOAT | REAL
       DOUBLE [PRECISION]|
       CHAR [(<n>)]| CHARACTER |
       STRING |
       VARCHAR [(<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>])
```

`<literal>`: See *section 3.1, What Is a Constant?*

`<succ_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 3.1, 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 of 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, classes of successor objects, inverse relationship and optional cardinality numbers. For example, the following statements define a relationship `directedBy` whose successor class is `movieDirector` and inverse relationship is `direct` defined in the `movieDirector` class:

```
CREATE CLASS movie (
    directedBy RELATIONSHIP (movieDirector)
        INVERSE movieDirector.direct,
    ...
);
CREATE CLASS movieDirector (
    direct RELATIONSHIP (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 RELATIONSHIP (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 RELATIONSHIP (movieDirector)
        CARDINALITY (0, 1)
        INVERSE movieDirector.direct,
    ...
);
```

By default, the successor objects of a relationship is ordered and keep their order as you add or remove successor objects. However, if the SET keyword is following RELATIONSHIP (or REFERENCES), the successor objects do not keep their order, but Matisse store those objects in any order for best performance. For example:

```
CREATE CLASS movie (
    directedBy RELATIONSHIP SET (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 RELATIONSHIP (tire)
        INVERSE tire.componentOf,
    ...
);
CREATE CLASS tire (
    componentOf READONLY RELATIONSHIP (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 each movie title unique:

```
CREATE CLASS movie (
    title STRING NOT NULL,
    CONSTRAINT movie_title_unique UNIQUE (title)
);
```

A unique constraint can use up to 256 characters when its attribute type is string. If a unique constraint uses more than one attribute and any of these attributes are string, you need to specify the maximum size for these string attributes. For example, if you want to make a 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)
);
```

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} class
         DROP { ATTRIBUTE attribute
              | {RELATIONSHIP | REFERENCES} relationship
              | {INHERIT | UNDER} <superclass>
              }
ALTER {CLASS | TABLE} class
         ADD { ATTRIBUTE attribute attribute_type
                 [DEFAULT literal] [NOT NULL]
             | {RELATIONSHIP | REFERENCES} relationship
                [[READONLY] RELATIONSHIP [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 | REFERENCES} relationship
                   [[READONLY] RELATIONSHIP [LIST | SET]]
                   ( succ_class [, ...] )
                   [CARDINALITY (min, max)]
                   INVERSE inv_class.inverse_relationship
               }
```

`attribute_type`: See *CREATE, on page 79*.

`literal`: See *section 3.1, 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 (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 (artist)
       CARDINALITY (0, 10)
       INVERSE artist.biography;
```

DROP

Syntax

```
DROP {CLASS | TABLE} class
```

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 movie;
```

Note that a `DROP CLASS` statement removes the attributes and relationships defined in the class unless they are used by other classes.

12.2 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 index ON class (
    attribute [ASC | DESC] [criterion_size]
    [, ...]
)
```

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.

For fixed size attributes, such as integers, `criterion_size` is not needed. For variable size attributes, such as character strings, `criterion_size` needs to be provided to specify how many characters, for example, to be used for indexing. If the indexed string is larger than the size defined for the criterion, it will be truncated in the index. The total size occupied by all the criteria must be less than 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 INDEX movieIndex ON movie (
    title ASC 32,
    runningTime ASC
);
```

DROP

Syntax

```
DROP INDEX index
```

Dropping Index

To remove an index, you can use `DROP INDEX` statement. The following statement removes the index `movieIndex`:

```
DROP INDEX movieIndex;
```

12.3 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
  entry_point_dictionary_name
  ON class (attribute)
  [CASE SENSITIVE]
  [MAKE_ENTRY make_entry_function];
```

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 entry_point_dictionary_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;
```

12.4 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 method_name
(<parameter_declaration [, ...]>)
RETURNS <data_type>
FOR class_name
BEGIN
  <statement>;
  [...]
END;
<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 the next release.

The data a method can return is either of attribute type, for examle INTEGER or DATE, or of object type such as class movie. If a method does not return anything, its return type is NULL.

Current limitation: A method cannot include UPDATE, INSERT, or DELETE statement. These types of statement in a method will be supported in the next release.

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:

```
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%');
```

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 method_name FOR class_name
```

Removing Method

A `DROP METHOD` statement removes a method defined for `class_name` from the database. For example:

```
DROP METHOD 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.

13 Manipulating Data

This section explains how to perform the following functions with SQL:

- Update attributes or relationships of objects
- Insert new objects into a database
- Delete some objects

These statements need to be executed within a transaction, not a version access.

13.1 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 [, ...]
    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';
```

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(top10Titles, next10Titles) INTERSECT kMovies
    WHERE week = DATE '2001-01-22';
```

The second statement above can also be stated as follows:

```
UPDATE boxOffice
    SET topTitlesForKids =
      (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(top10Titles, next10Titles) EXCEPT ncMovies
```

```
    WHERE week = DATE '2001-01-22';
```

13.2 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)
    VALUES (DATE '2001-01-29');
```

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);
```

13.3 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';
```

14 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 *12.4 Methods*.

14.1 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.

14.2 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 * 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 foo2()
RETURNS INTEGER
FOR Artist
BEGIN
  RETURN LENGTH(SELF.full_name());
END;
```

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.

14.3 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 *14.5 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;
```

14.4 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;
```

14.5 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.

15 Options

15.1 Setting Options

MAXOBJECTS

To limit the number of objects that are actually returned from server to client as a result of `SELECT` statement execution, you can use the `SET MAXOBJECTS` command. The syntax is as follows:

```
SET MAXOBJECTS number
```

where number is a positive integer value.

For example, the following statement sets the maximum number of objects returned from the server to 50:

```
SET MAXOBJECTS 50
```

After setting this option, the projection result shows only 50 objects even if a `SELECT` statement selects more than 50 objects. However, the information on the total number of objects selected by the `SQL` statement can be obtained through the *Matisse C-API for SQL*.

To revert to the default setting, i.e., no limit on the number of objects that are actually returned from server to client, use the following statement:

```
SET MAXOBJECTS OFF
```

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 Set<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;
};
```

Index

Symbols

- 38
% 45
* 38
+ 38
/ 38
< 37, 43
<= 37, 43
<> 37, 43
= 37, 43
> 37, 43
>= 37, 43
_ 46

A

AFTER 56
alias 29
ALL 40, 47, 51
ALTER 85
AND 30, 40
ANY 40, 47, 51
arithmetic expression 37
ASC 33
ASCII 44
assignment 106
AT 17
ATTRIBUTE 85
attribute 29
AVG 66, 70, 73

B

BEFORE 56
BETWEEN 40
boolean 16
bytes 18

C

CALL 99
CARDINALITY 79
CASE SENSITIVE 88
CAST 76
CHAR_LENGTH 63
character string constant 16
class statements 79–86
CLASS_ID 25
CLASS_NAME 25
COMMIT 59
COMMIT WORK 59
COMPILE 90
CONCAT 61
conditional join 27
constant 15
CONSTRAINT 79
CONTINUE 110
COUNT 54, 69, 71, 74
CREATE 79, 87, 88, 89
CURRENT_DATE 17, 75
CURRENT_TIMESTAMP 18, 75

D

DATE 17
date constant 17
DECLARE 108
DEFAULT 79
DefaultSelection 109
DELETE 96
DELETED 56
DESC 33
DISTINCT 33
division 39
DIVISION_BY_ZERO 111
DROP 86, 87, 88, 90
DROP SELECTION 28

E

ELEMENT 67
empty relationship 54
entry point 49
entry point dictionaries 88–89
ESCAPE 47
EXCEPT 94
exception 109
  handler 109
  unhandled 111
  user defined 111
EXIT 110
EXTRACT 76
  DAY 76
  HOUR 76
  MICROSECOND 76
  MINUTE 76
  MONTH 76
  SECOND 76
  YEAR 76

F

FALSE 16, 29
filter
  CLASS 53, 54
  ONLY 53, 54

FOR 103
FOREIGN KEY 80
FROM 22
full text search 49

G

GMT 17
GROUP BY 34

H

HAVING 35

I

identifier 19
IF 100
IN 48, 51
indices 87–88
INHERIT 80
INSERT 95
INSERTED 56
INSTR 62
interger constants 15
INTERSECT 94
INTO 28
INVERSE 79
IS NULL 41
ITERATE 105

J

join 26

K

keyword 19

L

LEAVE 104
LENGTH 63
LIKE 46
LIST 79
LIST 70
list functions 66–70
LOCAL 17
LOOP 101
LOWER 63
LTRIM 63

M

MAX 67, 71, 74
METHOD 89
Method Invocation 98
Methods 89
MIN 68, 72, 74

N

natural join 26
navigational queries 52
NOT 32, 40, 41
NULL 42, 51, 54
Null 38
null value 15
numeric constants 15

O

OID 24
ONLY 22
operator
  arithmetic 37
  comparison 37
  negation 39
OR 30
ORDER BY 33

P

pass by reference 106
pattern 45
precedence 31
predecessor 51
predicate 29
  BETWEEN 40
  entry point 49

IN 48, 51
IS OF 32
LIKE 46
NULL 41
PRIMARY KEY 79
Projection 23

R

READ ONLY 58
READ WRITE 59
real constants 16
REF 24
REFERENCES 79
REFERENCES 80
Referential Constraint 84
RELATIONSHIP 79
relationship 51
    navigation 52

REPEAT 102
RESIGNAL 107
result types 38
RETURN 105
RETURNING 96
ROLLBACK 60
ROLLBACK WORK 60
RTRIM 64

S

search criteria 29
SELECT 22
Selection 27
SELF 99
SET 79
SET 106
set functions 70–72
SET TRANSACTION 59
SIGNAL 107
SQL functions 61–66
statement block 107
Static Method 99
string 16
SUBLIST 68
SUBSTR 65
SUBSTRING 65
successor 51
SUM 69, 72, 75

T

text comparison 43
TIMESTAMP 17
timestamp constant 17
TRANSACTION 58
TRUE 16, 29
type compatibility 106
type predicate 32

U

UNDER 80
UNION 94
UNIQUE 87, 88
UNIQUE 79
Unique Constraint 84
UNKNOWN 29, 43
UPDATE 92
UPDATED 56
UPPER 65
UTC 17

V

version 56
version travel 56

W

WHERE 29
WHILE 102
wildcard character 45