

FileMaker® 15

SQL Reference



© 2013-2016 FileMaker, Inc. All Rights Reserved.

FileMaker, Inc.
5201 Patrick Henry Drive
Santa Clara, California 95054

FileMaker and FileMaker Go are trademarks of FileMaker, Inc. registered in the U.S. and other countries. The file folder logo and FileMaker WebDirect are trademarks of FileMaker, Inc. All other trademarks are the property of their respective owners. FileMaker documentation is copyrighted. You are not authorized to make additional copies or distribute this documentation without written permission from FileMaker. You may use this documentation solely with a valid licensed copy of FileMaker software.

All persons, companies, email addresses, and URLs listed in the examples are purely fictitious and any resemblance to existing persons, companies, email addresses, or URLs is purely coincidental. Credits are listed in the Acknowledgments documents provided with this software. Mention of third-party products and URLs is for informational purposes only and constitutes neither an endorsement nor a recommendation. FileMaker, Inc. assumes no responsibility with regard to the performance of these products.

For more information, visit our website at <http://www.filemaker.com>.

Edition: 01

Contents

Chapter 1

Introduction

About this reference	5
Where to find FileMaker documentation	5
About SQL	5
Using a FileMaker database as a data source	6
Using the ExecuteSQL function	6

Chapter 2

Supported standards

Support for Unicode characters	7
SQL statements	7
SELECT statement	8
SQL clauses	9
FROM clause	9
WHERE clause	10
GROUP BY clause	11
HAVING clause	11
UNION operator	12
ORDER BY clause	12
OFFSET and FETCH FIRST clauses	13
FOR UPDATE clause	14
DELETE statement	17
INSERT statement	17
UPDATE statement	19
CREATE TABLE statement	20
TRUNCATE TABLE statement	21
ALTER TABLE statement	22
CREATE INDEX statement	22
DROP INDEX statement	23
SQL expressions	23
Field names	23
Constants	24
Exponential/scientific notation	25
Numeric operators	25
Character operators	25
Date operators	25
Relational operators	26
Logical operators	27
Operator precedence	28

SQL functions	29
Aggregate functions	29
Functions that return character strings	30
Functions that return numbers	31
Functions that return dates	33
Conditional functions	33
FileMaker system objects	34
FileMaker system tables	34
FileMaker system columns	35
Reserved SQL keywords	36
<i>Index</i>	39

Chapter 1

Introduction

As a database developer, you can use FileMaker Pro to create database solutions without any knowledge of SQL. But if you have some knowledge of SQL, you can use a FileMaker database file as an ODBC or JDBC data source, sharing your data with other applications using ODBC and JDBC. You can also use the FileMaker Pro ExecuteSQL function to retrieve data from any table occurrence within a FileMaker Pro database.

This reference describes the SQL statements and standards supported by FileMaker. The FileMaker ODBC and JDBC client drivers support all of the SQL statements described in this reference. The FileMaker Pro ExecuteSQL function supports only the SELECT statement.

About this reference

- For information on using ODBC and JDBC with previous versions of FileMaker Pro, see <http://www.filemaker.com/documentation>.
- This reference assumes that you are familiar with the basics of using FileMaker Pro functions, coding ODBC and JDBC applications, and constructing SQL queries. Refer to a third-party book for more information on these topics.
- This reference uses “FileMaker Pro” to refer to both FileMaker Pro and FileMaker Pro Advanced, unless describing specific FileMaker Pro Advanced features.

Where to find FileMaker documentation

- In FileMaker Pro, choose **Help** menu > **Product Documentation**.
- In FileMaker Server Admin Console, choose **Help** menu > **FileMaker Server Product Documentation**.
- To learn about, view, or download additional FileMaker documentation, visit <http://www.filemaker.com/documentation>.

About SQL

SQL, or Structured Query Language, is a programming language that was designed to query data from a relational database. The primary statement used to query a database is the SELECT statement.

In addition to language for querying a database, SQL provides statements for performing data manipulation, which allow you to add, update, and delete data.

SQL also provides statements for performing data definition. These statements allow you to create and modify tables and indexes.

The SQL statements and standards supported by FileMaker are described in chapter 2, “Supported standards.”

Using a FileMaker database as a data source

When you host a FileMaker database as an ODBC or JDBC data source, FileMaker data can be shared with ODBC- and JDBC-compliant applications. The applications connect to the FileMaker data source using the FileMaker client drivers, construct and execute the SQL queries using ODBC or JDBC, and process the data retrieved from the FileMaker database solution.

See [FileMaker ODBC and JDBC Guide](#) for extensive information on how you can use FileMaker software as a data source for ODBC and JDBC applications.

The FileMaker ODBC and JDBC client drivers support all of the SQL statements described in this reference.

Using the ExecuteSQL function

The FileMaker Pro ExecuteSQL function lets you retrieve data from table occurrences named in the relationships graph but independent of any defined relationships. You can retrieve data from multiple tables without creating table joins or any relationship between the tables. In some cases, you may be able to reduce the complexity of your relationships graph by using the ExecuteSQL function.

The fields you query with the ExecuteSQL function do not have to be on any layout, so you can use the ExecuteSQL function to retrieve data independent of any layout context. Because of this context independence, using the ExecuteSQL function in scripts may improve the portability of the scripts. You can use the ExecuteSQL function anywhere you can specify calculations, including for charting and reporting.

The ExecuteSQL function supports only the SELECT statement, described in the section “SELECT statement” on page 8.

Also, the ExecuteSQL function accepts only the SQL-92 syntax ISO date and time formats with no braces ({}). The ExecuteSQL function does not accept the ODBC/JDBC format date, time, and timestamp constants in braces.

For information about the syntax and use of the ExecuteSQL function, see [FileMaker Pro Help](#).

Chapter 2

Supported standards

This reference describes the SQL statements and constructs supported by FileMaker. The FileMaker ODBC and JDBC client drivers support all of the SQL statements described in this chapter. The FileMaker Pro ExecuteSQL function supports only the SELECT statement.

Use the client drivers to access a FileMaker database solution from an ODBC- or JDBC-compliant application. The FileMaker database solution can be hosted by either FileMaker Pro or FileMaker Server.

- The ODBC client driver supports ODBC 3.0 Level 1.
- The JDBC client driver provides partial support for the JDBC 3.0 specification.
- The ODBC and JDBC client drivers both support SQL-92 entry-level conformance, with some SQL-92 intermediate features.

Support for Unicode characters

The ODBC and JDBC client drivers support the Unicode API. However, if you're creating a custom application that uses the client drivers, use ASCII for field names, table names, and filenames (in case a non-Unicode query tool or application is used).

Note To insert and retrieve Unicode data, use `SQL_C_WCHAR`.

SQL statements

The ODBC and JDBC client drivers provide support for the following SQL statements:

- SELECT (page 8)
- DELETE (page 17)
- INSERT (page 17)
- UPDATE (page 19)
- CREATE TABLE (page 20)
- TRUNCATE TABLE (page 21)
- ALTER TABLE (page 22)
- CREATE INDEX (page 22)
- DROP INDEX (page 23)

The client drivers also support FileMaker data type mapping to ODBC SQL and JDBC SQL data types. See [FileMaker ODBC and JDBC Guide](#) for data type conversions. For more information on constructing SQL queries, refer to a third-party book.

Note The ODBC and JDBC client drivers do not support FileMaker portals.

SELECT statement

Use the `SELECT` statement to specify which columns you're requesting. Follow the `SELECT` statement with the column expressions (similar to field names) you want to retrieve (for example, `last_name`). Expressions can include mathematical operations or string manipulation (for example, `SALARY * 1.05`).

The `SELECT` statement can use a variety of clauses:

```
SELECT [DISTINCT] { * | column_expression [[AS] column_alias], ... }
FROM table_name [table_alias], ...
[ WHERE expr1 rel_operator expr2 ]
[ GROUP BY {column_expression, ...} ]
[ HAVING expr1 rel_operator expr2 ]
[ UNION [ALL] (SELECT...) ]
[ ORDER BY {sort_expression [DESC | ASC]}, ... ]
[ OFFSET n {ROWS | ROW} ]
[ FETCH FIRST [ n [ PERCENT ] ] { ROWS | ROW } { ONLY | WITH TIES } ]
[ FOR UPDATE [OF {column_expression, ...}] ]
```

Items in brackets are optional.

`column_alias` can be used to give the column a more descriptive name, or to abbreviate a longer column name. For example, to assign the alias `department` to the column `dept`:

```
SELECT dept AS department FROM emp
```

Field names can be prefixed with the table name or the table alias. For example, `EMP.LAST_NAME` or `E.LAST_NAME`, where `E` is the alias for the table `EMP`.

The `DISTINCT` operator can precede the first column expression. This operator eliminates duplicate rows from the result of a query. For example:

```
SELECT DISTINCT dept FROM emp
```


SQL clauses

The ODBC and JDBC client drivers provide support for the following SQL clauses.

Use this SQL clause	To
FROM (page 9)	Indicate which tables are used in the <code>SELECT</code> statement.
WHERE (page 10)	Specify the conditions that records must meet to be retrieved (like a FileMaker Pro find request).
GROUP BY (page 11)	Specify the names of one or more fields by which the returned values should be grouped. This clause is used to return a set of aggregate values by returning one row for each group (like a FileMaker Pro subsummary).
HAVING (page 11)	Specify conditions for groups of records (for example, display only the departments that have salaries totaling more than \$200,000).
UNION (page 12)	Combine the results of two or more <code>SELECT</code> statements into a single result.
ORDER BY (page 12)	Indicate how the records are sorted.
OFFSET (page 13)	State the number of rows to be skipped before starting to retrieve rows.
FETCH FIRST (page 13)	Specify the number of rows to be retrieved. No more than the specified number of rows are returned although fewer rows may be returned if the query yields less than the number of rows specified.
FOR UPDATE (page 14)	Perform Positioned Updates or Positioned Deletes via SQL cursors.

Note If you attempt to retrieve data from a table with no columns, the `SELECT` statement returns nothing.

FROM clause

The `FROM` clause indicates the tables that are used in the `SELECT` statement. The format is:

```
FROM table_name [table_alias] [, table_name [table_alias]]
```

`table_name` is the name of a table in the current database. The table name must begin with an alphabetic character. If the table name begins with other than an alphabetic character, enclose it in double quotation marks (quoted identifier).

`table_alias` can be used to give the table a more descriptive name, to abbreviate a longer table name, or to include the same table in the query more than once (for example, in self-joins).

Field names begin with an alphabetic character. If the field name begins with other than an alphabetic character, enclose it in double quotation marks (quoted identifier). For example, the `ExecuteSQL` statement for the field named `_LASTNAME` is:

```
SELECT "_LASTNAME" from emp
```

Field names can be prefixed with the table name or the table alias. For example, given the table specification `FROM employee E`, you can refer to the `LAST_NAME` field as `E.LAST_NAME`. Table aliases must be used if the `SELECT` statement joins a table to itself. For example:

```
SELECT * FROM employee E, employee F WHERE E.manager_id =
F.employee_id
```

The equal sign (=) includes only matching rows in the results.

If you are joining more than one table, and you want to discard all rows that don't have corresponding rows in both source tables, you can use `INNER JOIN`. For example:

```
SELECT *
  FROM Salespeople INNER JOIN Sales_Data
  ON Salespeople.Salesperson_ID = Sales_Data.Salesperson_ID
```

If you are joining two tables, but you don't want to discard rows of the first table (the "left" table), you can use `LEFT OUTER JOIN`.

```
SELECT *
  FROM Salespeople LEFT OUTER JOIN Sales_Data
  ON Salespeople.Salesperson_ID = Sales_Data.Salesperson_ID
```

Every row from the "Salespeople" table will appear in the joined table.

Notes

- `RIGHT OUTER JOIN` is not currently supported.
- `FULL OUTER JOIN` is not currently supported.

WHERE clause

The `WHERE` clause specifies the conditions that records must meet to be retrieved. The `WHERE` clause contains conditions in the form:

```
WHERE expr1 rel_operator expr2
```

`expr1` and `expr2` can be field names, constant values, or expressions.

`rel_operator` is the relational operator that links the two expressions. For example, the following `SELECT` statement retrieves the names of employees who make \$20,000 or more.

```
SELECT last_name,first_name FROM emp WHERE salary >= 20000
```

The `WHERE` clause can also use expressions such as these:

```
WHERE expr1 IS NULL
WHERE NOT expr2
```

Note If you use fully qualified names in the `SELECT` (projection) list, you must also use fully qualified names in the related `WHERE` clause.

GROUP BY clause

The `GROUP BY` clause specifies the names of one or more fields by which the returned values should be grouped. This clause is used to return a set of aggregate values. It has the following format:

```
GROUP BY columns
```

The scope of the `GROUP BY` clause is the table expression in the `FROM` clause. As a result, the column expressions specified by `columns` must be from the tables specified in the `FROM` clause. A column expression can be one or more field names of the database table separated by commas.

Example

The following example sums the salaries in each department.

```
SELECT dept_id, SUM (salary) FROM emp GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

HAVING clause

The `HAVING` clause enables you to specify conditions for groups of records (for example, display only the departments that have salaries totaling more than \$200,000). It has the following format:

```
HAVING expr1 rel_operator expr2
```

`expr1` and `expr2` can be field names, constant values, or expressions. These expressions do not have to match a column expression in the `SELECT` clause.

`rel_operator` is the relational operator that links the two expressions.

Example

The following example returns only the departments whose sums of salaries are greater than \$200,000:

```
SELECT dept_id, SUM (salary) FROM emp  
GROUP BY dept_id HAVING SUM (salary) > 200000
```

UNION operator

The `UNION` operator combines the results of two or more `SELECT` statements into a single result. The single result is all of the returned records from the `SELECT` statements. By default, duplicate records are not returned. To return duplicate records, use the `ALL` keyword (`UNION ALL`). The format is:

```
SELECT statement UNION [ALL] SELECT statement
```

When using the `UNION` operator, the select lists for each `SELECT` statement must have the same number of column expressions, with the same data types, and must be specified in the same order. For example:

```
SELECT last_name, salary, hire_date FROM emp UNION SELECT name, pay,
birth_date FROM person
```

This example has the same number of column expressions, and each column expression, in order, has the same data type.

The following example is not valid because the data types of the column expressions are different (`SALARY` from `EMP` has a different data type than `LAST_NAME` from `RAISES`). This example has the same number of column expressions in each `SELECT` statement, but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp UNION SELECT salary, last_name FROM
raises
```

ORDER BY clause

The `ORDER BY` clause indicates how the records are to be sorted. If your `SELECT` statement doesn't include an `ORDER BY` clause, the records may be returned in any order.

The format is:

```
ORDER BY {sort_expression [DESC | ASC]}, ...
```

`sort_expression` can be the field name or the positional number of the column expression to use. The default is to perform an ascending (`ASC`) sort.

For example, to sort by `last_name` then by `first_name`, you could use either of the following `SELECT` statements:

```
SELECT emp_id, last_name, first_name FROM emp ORDER BY last_name,
first_name
```

or

```
SELECT emp_id, last_name, first_name FROM emp ORDER BY 2,3
```

The second example uses the positional numbers 2 and 3 to get the same ordering as the prior example that specified `last_name` and `first_name` explicitly.

Note FileMaker SQL uses the Unicode binary sort order, which is different from the FileMaker Pro sort order used with language sorting or with the default language-neutral sort order.

OFFSET and FETCH FIRST clauses

The `OFFSET` and `FETCH FIRST` clauses are used to return a specified range of rows beginning from a particular starting point in a result set. The ability to limit the rows retrieved from large result sets allows you to “page” through the data and improves efficiency.

The `OFFSET` clause indicates the number of rows to skip before starting to return data. If the `OFFSET` clause is not used in a `SELECT` statement, the starting row is 0. The `FETCH FIRST` clause specifies the number of rows to be returned, either as an unsigned integer greater than or equal to 1 or as a percentage, from the starting point indicated in the `OFFSET` clause. If both `OFFSET` and `FETCH FIRST` are used in a `SELECT` statement, the `OFFSET` clause should come first.

The `OFFSET` and `FETCH FIRST` clauses are not supported in subqueries.

OFFSET format

The `OFFSET` format is:

```
OFFSET n {ROWS | ROW} ]
```

`n` is an unsigned integer. If `n` is larger than the number of rows returned in the result set, then nothing is returned and no error message appears.

`ROWS` is the same as `ROW`.

FETCH FIRST format

The `FETCH FIRST` format is:

```
FETCH FIRST [ n [ PERCENT ] ] { ROWS | ROW } { ONLY | WITH TIES } ]
```

`n` is the number of rows to be returned. The default value is 1 if `n` is omitted.

`n` is an unsigned integer greater than or equal to 1 unless it is followed by `PERCENT`. If `n` is followed by `PERCENT`, the value may be either a positive fractional value or an unsigned integer.

`ROWS` is the same as `ROW`.

`WITH TIES` must be used with the `ORDER BY` clause.

`WITH TIES` allows more rows to be returned than specified in the `FETCH` count value because peer rows, those rows that are not distinct based on the `ORDER BY` clause, are also returned.

Examples

For example, to return information from the twenty-sixth row of the result set sorted by `last_name` then by `first_name`, use the following `SELECT` statement:

```
SELECT emp_id, last_name, first_name FROM emp ORDER BY last_name,
first_name OFFSET 25 ROWS
```

To specify that you want to return only ten rows:

```
SELECT emp_id, last_name, first_name FROM emp ORDER BY last_name,
first_name OFFSET 25 ROWS FETCH FIRST 10 ROWS ONLY
```

To return the ten rows and their peer rows (rows that are not distinct based on the `ORDER BY` clause):

```
SELECT emp_id, last_name, first_name FROM emp ORDER BY last_name,  
first_name OFFSET 25 ROWS FETCH FIRST 10 ROWS WITH TIES
```

FOR UPDATE clause

The `FOR UPDATE` clause locks records for Positioned Updates or Positioned Deletes via SQL cursors. The format is:

```
FOR UPDATE [OF column_expressions]
```

`column_expressions` is a list of field names in the database table that you intend to update, separated by a comma. `column_expressions` is optional, and is ignored.

Example

The following example returns all records in the employee database that have a `SALARY` field value of more than \$20,000. When each record is fetched, it is locked. If the record is updated or deleted, the lock is held until you commit the change. Otherwise, the lock is released when you fetch the next record.

```
SELECT * FROM emp WHERE salary > 20000  
FOR UPDATE OF last_name, first_name, salary
```

Additional examples:

Using	Sample SQL
text constant	<code>SELECT 'CatDog' FROM Salespeople</code>
numeric constant	<code>SELECT 999 FROM Salespeople</code>
date constant	<code>SELECT DATE '2016-06-05' FROM Salespeople</code>
time constant	<code>SELECT TIME '02:49:03' FROM Salespeople</code>
timestamp constant	<code>SELECT TIMESTAMP '2016-06-05 02:49:03' FROM Salespeople</code>
text column	<code>SELECT Company_Name FROM Sales_Data</code> <code>SELECT DISTINCT Company_Name FROM Sales_Data</code>
numeric column	<code>SELECT Amount FROM Sales_Data</code> <code>SELECT DISTINCT Amount FROM Sales_Data</code>
date column	<code>SELECT Date_Sold FROM Sales_Data</code> <code>SELECT DISTINCT Date_Sold FROM Sales_Data</code>
time column	<code>SELECT Time_Sold FROM Sales_Data</code> <code>SELECT DISTINCT Time_Sold FROM Sales_Data</code>
timestamp column	<code>SELECT Timestamp_Sold FROM Sales_Data</code> <code>SELECT DISTINCT Timestamp_Sold FROM Sales_Data</code>
BLOB ^a column	<code>SELECT Company_Brochures FROM Sales_Data</code> <code>SELECT GETAS(Company_Logo, 'JPEG') FROM Sales_Data</code>
Wildcard *	<code>SELECT * FROM Salespeople</code> <code>SELECT DISTINCT * FROM Salespeople</code>

a. A BLOB is a FileMaker database file container field.

Notes from the examples

A `column` is a reference to a field in the FileMaker database file. (The field can contain many distinct values.)

The asterisk (*) wildcard character is shorthand for “everything”. For the example `SELECT * FROM Salespeople`, the result is all the columns in the `Salespeople` table. For the example `SELECT DISTINCT * FROM Salespeople`, the result is all the unique rows in the `Salespeople` table (no duplicates).

- FileMaker does not store data for empty strings, so the following queries always return no records:

```
SELECT * FROM test WHERE c = ''
SELECT * FROM test WHERE c <> ''
```

- If you use `SELECT` with binary data, you must use the `GetAs()` function to specify the stream to return. See the following section “Retrieving the contents of a container field: `CAST()` function and `GetAs()` function,” for more information.

Retrieving the contents of a container field: CAST() function and GetAs() function

You can retrieve binary data, file reference information, or data of a specific file type from a container field.

If file data or JPEG binary data exists, the `SELECT` statement with `GetAS(field name, 'JPEG')` retrieves the data in binary form; otherwise, the `SELECT` statement with field name returns `NULL`.

To retrieve file reference information from a container field, such as the file path to a file, picture, or QuickTime movie, use the `CAST()` function with a `SELECT` statement. For example:

```
SELECT CAST(Company_Brochures AS VARCHAR) FROM Sales_Data
```

In this example, if you:

- Inserted a file into the container field using FileMaker Pro but stored only a reference to the file, the `SELECT` statement retrieves the file reference information as type `SQL_VARCHAR`.
- Inserted the contents of a file into the container field using FileMaker Pro, the `SELECT` statement retrieves the name of the file.
- Imported a file into the container field from another application, the `SELECT` statement displays '?' (the file displays as **Untitled.dat** in FileMaker Pro).

To retrieve data from a container field, use the `GetAs()` function. You may use the `DEFAULT` option or specify the file type. The `DEFAULT` option retrieves the master stream for the container without the need to explicitly define the stream type:

```
SELECT GetAs(Company_Brochures, DEFAULT) FROM Sales_Data
```

To retrieve an individual stream type from a container, use the `GetAs()` function with the file's type based on how the data was inserted into the container field in FileMaker Pro. For example:

- If the data was inserted using the **Insert > File** command, specify `'FILE'` in the `GetAs()` function. For example:

```
SELECT GetAs(Company_Brochures, 'FILE') FROM Sales_Data
```

- If the data was inserted using the **Insert > Sound** command (Standard sound — MAC OS X raw format), specify `'snd'` in the `GetAs()` function. For example:

```
SELECT GetAs(Company_Meeting, 'snd ') FROM Company_Newsletter
```


- If the data was inserted using the **Insert > Picture** command, drag and drop, or paste from the clipboard, specify one of the file types listed in the following table. For example:

```
SELECT GetAs(Company_Logo, 'JPEG') FROM Company_Icons
```

File type	Description	File type	Description
'GIFf'	Graphics Interchange Format	'PNTG'	MacPaint
'JPEG'	Photographic images	' .SGI'	Generic bitmap format
'JP2 '	JPEG 2000	'TIFF'	Raster file format for digital images
'PDF '	Portable Document Format	'TPIC'	Targa
'PNGf'	Bitmap image format	'8BPS'	PhotoShop (PSD)

DELETE statement

Use the `DELETE` statement to delete records from a database table. The format of the `DELETE` statement is:

```
DELETE FROM table_name [ WHERE { conditions } ]
```

Note The `WHERE` clause determines which records are to be deleted. If you don't include the `WHERE` keyword, all records in the table are deleted (but the table is left intact).

Example

An example of a `DELETE` statement on the `Employee` table is:

```
DELETE FROM emp WHERE emp_id = 'E10001'
```

Each `DELETE` statement removes every record that meets the conditions in the `WHERE` clause. In this case, every record having the employee ID `E10001` is deleted. Because employee IDs are unique in the `Employee` table, only one record is deleted.

INSERT statement

Use the `INSERT` statement to create records in a database table. You can specify either:

- A list of values to be inserted as a new record
- A `SELECT` statement that copies data from another table to be inserted as a set of new records

The format of the `INSERT` statement is:

```
INSERT INTO table_name [(column_name, ...)] VALUES (expr, ...)
```

`column_name` is an optional list of column names that provides the name and order of the columns whose values are specified in the `VALUES` clause. If you omit `column_name`, the value expressions (`expr`) must provide values for all columns defined in the table and must be in the same order that the columns are defined for the table. `column_name` may also specify a field repetition, for example `lastDates[4]`.

`expr` is the list of expressions giving the values for the columns of the new record. Usually the expressions are constant values for the columns (but they can also be a subquery). You must enclose character string values in pairs of single quotation marks ('). To include a single quotation mark in a character string value enclosed by single quotation marks, use two single quotation marks together (for example, 'Don' 't').

Subqueries must be enclosed in parentheses.

The following example inserts a list of expressions:

```
INSERT INTO emp (last_name, first_name, emp_id, salary, hire_date)
VALUES ('Smith', 'John', 'E22345', 27500, DATE '2016-06-05')
```

Each `INSERT` statement adds one record to the database table. In this case a record has been added to the employee database table, `EMP`. Values are specified for five columns. The remaining columns in the table are assigned a blank value, meaning Null.

Note In container fields, you can `INSERT` text only, unless you prepare a parameterized statement and stream the data from your application. To use binary data, you may simply assign the filename by enclosing it in single quotation marks or use the `PutAs()` function. When specifying the filename, the file type is deduced from the file extension:

```
INSERT INTO table_name (container_name) VALUES(? AS 'filename.file
extension')
```

Unsupported file types will be inserted as type `FILE`.

When using the `PutAs()` function, specify the type: `PutAs(col, 'type')`, where the type value is a supported file type as described in “Retrieving the contents of a container field: `CAST()` function and `GetAs()` function” on page 16.

The `SELECT` statement is a query that returns values for each `column_name` value specified in the column name list. Using a `SELECT` statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single `INSERT` statement.

Here's an example of an `INSERT` statement that uses a `SELECT` statement:

```
INSERT INTO emp1 (first_name, last_name, emp_id, dept, salary)
SELECT first_name, last_name, emp_id, dept, salary from emp
WHERE dept = 'D050'
```

In this type of `INSERT` statement, the number of columns to be inserted must match the number of columns in the `SELECT` statement. The list of columns to be inserted must correspond to the columns in the `SELECT` statement just as it would to a list of value expressions in the other type of `INSERT` statement. For example, the first column inserted corresponds to the first column selected; the second inserted to the second, and so on.

The size and data type of these corresponding columns must be compatible. Each column in the `SELECT` list should have a data type that the ODBC or JDBC client driver accepts on a regular `INSERT/UPDATE` of the corresponding column in the `INSERT` list. Values are truncated when the size of the value in the `SELECT` list column is greater than the size of the corresponding `INSERT` list column.

The `SELECT` statement is evaluated before any values are inserted.

UPDATE statement

Use the `UPDATE` statement to change records in a database table. The format of the `UPDATE` statement is:

```
UPDATE table_name SET column_name = expr, ... [ WHERE { conditions } ]
```

`column_name` is the name of a column whose value is to be changed. Several columns can be changed in one statement.

`expr` is the new value for the column.

Usually the expressions are constant values for the columns (but they can also be a subquery). You must enclose character string values in pairs of single quotation marks (`'`). To include a single quotation mark in a character string value enclosed by single quotation marks, use two single quotation marks together (for example, `'Don' 't'`).

Subqueries must be enclosed in parentheses.

The `WHERE` clause is any valid clause. It determines which records are updated.

Examples

An example of an `UPDATE` statement on the `Employee` table is:

```
UPDATE emp SET salary=32000, exempt=1 WHERE emp_id = 'E10001'
```

The `UPDATE` statement changes every record that meets the conditions in the `WHERE` clause. In this case the salary and exempt status are changed for all employees having the employee ID `E10001`. Because employee IDs are unique in the `Employee` table, only one record is updated. Here's an example using a subquery:

```
UPDATE emp SET salary = (SELECT avg(salary) from emp) WHERE emp_id = 'E10001'
```

In this case, the salary is changed to the average salary in the company for the employee having employee ID `E10001`.

Note In container fields, you can `UPDATE` with text only, unless you prepare a parameterized statement and stream the data from your application. To use binary data, you may simply assign the filename by enclosing it in single quotation marks or use the `PutAs()` function. When specifying the filename, the file type is deduced from the file extension:

```
UPDATE table_name SET (container_name) = ? AS 'filename.file extension'
```

Unsupported file types will be inserted as type `FILE`.

When using the `PutAs()` function, specify the type: `PutAs(col, 'type')`, where the type value is a supported file type as described in “Retrieving the contents of a container field: `CAST()` function and `GetAs()` function” on page 16.

CREATE TABLE statement

Use the `CREATE TABLE` statement to create a table in a database file. The format of the `CREATE TABLE` statement is:

```
CREATE TABLE table_name ( table_element_list [,
table_element_list... ] )
```

Within the statement, you specify the name and data type of each column.

- `table_name` is the name of the table. `table_name` has a 100 character limit. A table with the same name must not already be defined. The table name must begin with an alphabetic character. If the table name begins with other than an alphabetic character, enclose it in double quotation marks (quoted identifier).
- The format for `table_element_list` is:

```
field_name field_type [[repetitions]]
[DEFAULT expr] [UNIQUE | NOT NULL | PRIMARY KEY | GLOBAL]
[EXTERNAL relative_path_string [SECURE | OPEN calc_path_string]]
```

- `field_name` is the name of the field. Field names must be unique. Field names begin with an alphabetic character. If the field name begins with other than an alphabetic character, enclose it in double quotation marks (quoted identifier). For example, the `CREATE TABLE` statement for the field named `_LASTNAME` is:

```
CREATE TABLE "_EMPLOYEE" (ID INT PRIMARY KEY, "_FIRSTNAME"
VARCHAR(20), "_LASTNAME" VARCHAR(20))
```

- For the `CREATE TABLE` statement `repetitions`, specify a field repetition by using a number from 1 to 32000 in brackets after the field type. For example:

```
EMPLOYEE_ID INT[4]
LASTNAME VARCHAR(20)[4]
```

- `field_type` may be any of the following: `NUMERIC`, `DECIMAL`, `INT`, `DATE`, `TIME`, `TIMESTAMP`, `VARCHAR`, `CHARACTER VARYING`, `BLOB`, `VARBINARY`, `LONGVARBINARY`, or `BINARY VARYING`. For `NUMERIC` and `DECIMAL`, you can specify the precision and scale. For example: `DECIMAL(10,0)`. For `TIME` and `TIMESTAMP`, you can specify the precision. For example: `TIMESTAMP(6)`. For `VARCHAR` and `CHARACTER VARYING`, you can specify the length of the string. For example: `VARCHAR(255)`.
- The `DEFAULT` keyword allows you to set a default value for a column. For `expr`, you may use a constant value or expression. Allowable expressions are `USER`, `USERNAME`, `CURRENT_USER`, `CURRENT_DATE`, `CURDATE`, `CURRENT_TIME`, `CURTIME`, `CURRENT_TIMESTAMP`, `CURTIMESTAMP`, and `NULL`.
- Defining a column to be `UNIQUE` automatically selects the **Unique Validation Option** for the corresponding field in the FileMaker database file.
- Defining a column to be `NOT NULL` automatically selects the **Not Empty Validation Option** for the corresponding field in the FileMaker database file. The field is flagged as a **Required Value** in the **Fields** tab of the Manage Database dialog box in FileMaker Pro.

- To define a column as a container field, use `BLOB`, `VARBINARY`, or `BINARY VARYING` for the `field_type`.
- To define a column as a container field that stores data externally, use the `EXTERNAL` keyword. The `relative_path_string` defines the folder where the data is stored externally, relative to the location of the FileMaker database. This path must be specified as the base directory in the FileMaker Pro Manage Containers dialog box. You must specify either `SECURE` for secure storage or `OPEN` for open storage. If you are using open storage, the `calc_path_string` is the folder inside the `relative_path_string` folder where container objects are to be stored. The path must use forward slashes (/) in the folder name.

Examples

Using	Sample SQL
text column	<code>CREATE TABLE T1 (C1 VARCHAR, C2 VARCHAR (50), C3 VARCHAR (1001), C4 VARCHAR (500276))</code>
text column, NOT NULL	<code>CREATE TABLE T1NN (C1 VARCHAR NOT NULL, C2 VARCHAR (50) NOT NULL, C3 VARCHAR (1001) NOT NULL, C4 VARCHAR (500276) NOT NULL)</code>
numeric column	<code>CREATE TABLE T2 (C1 DECIMAL, C2 DECIMAL (10,0), C3 DECIMAL (7539,2), C4 DECIMAL (497925,301))</code>
date column	<code>CREATE TABLE T3 (C1 DATE, C2 DATE, C3 DATE, C4 DATE)</code>
time column	<code>CREATE TABLE T4 (C1 TIME, C2 TIME, C3 TIME, C4 TIME)</code>
timestamp column	<code>CREATE TABLE T5 (C1 TIMESTAMP, C2 TIMESTAMP, C3 TIMESTAMP, C4 TIMESTAMP)</code>
column for container field	<code>CREATE TABLE T6 (C1 BLOB, C2 BLOB, C3 BLOB, C4 BLOB)</code>
column for external storage	<code>CREATE TABLE T7 (C1 BLOB EXTERNAL 'Files/MyDatabase/' SECURE)</code>
column for container field	<code>CREATE TABLE T8 (C1 BLOB EXTERNAL 'Files/MyDatabase/' OPEN 'Objects')</code>

TRUNCATE TABLE statement

Use the `TRUNCATE TABLE` statement to quickly delete all records in the specified table, emptying the table of all data.

```
TRUNCATE TABLE table_name
```

You cannot specify a `WHERE` clause with the `TRUNCATE TABLE` statement. The `TRUNCATE TABLE` statement deletes all records.

Only the records in the table specified by `table_name` are deleted. Records from any related tables are not affected.

The `TRUNCATE TABLE` statement needs to be able to lock all records in the table in order to delete the record data. If any record in the table is locked by another user, FileMaker returns the error code 301 (“Record is in use by another user”).

ALTER TABLE statement

Use the `ALTER TABLE` statement to change the structure of an existing table in a database file. You can modify only one column in each statement. The formats of the `ALTER TABLE` statement are:

```
ALTER TABLE table_name ADD [COLUMN] column_definition
```

```
ALTER TABLE table_name DROP [COLUMN] unqualified_column_name
```

```
ALTER TABLE table_name ALTER [COLUMN] column_definition SET DEFAULT
expr
```

```
ALTER TABLE table_name ALTER [COLUMN] column_definition DROP DEFAULT
```

You must know the table's structure and how you want to modify it before using the `ALTER TABLE` statement.

Examples

To	Sample SQL
add columns	<code>ALTER TABLE Salespeople ADD C1 VARCHAR</code>
remove columns	<code>ALTER TABLE Salespeople DROP C1</code>
set the default value for a column	<code>ALTER TABLE Salespeople ALTER Company SET DEFAULT 'FileMaker'</code>
remove the default value for a column	<code>ALTER TABLE Salespeople ALTER Company DROP DEFAULT</code>

Note `SET DEFAULT` and `DROP DEFAULT` do not affect existing rows in the table, but change the default value for rows that are subsequently added to the table.

CREATE INDEX statement

Use the `CREATE INDEX` statement to speed searches in your database file. The format of the `CREATE INDEX` statement is:

```
CREATE INDEX ON table_name.column_name
CREATE INDEX ON table_name (column_name)
```

`CREATE INDEX` is supported for a single column (multi-column indexes are not supported). Indexes are not allowed on columns that correspond to container field types, summary fields, fields that have the global storage option, or unstored calculation fields in a FileMaker database file.

Creating an index for a text column automatically selects the Storage Option of **Minimal in Indexing** for the corresponding field in the FileMaker database file. Creating an index for a non-text column (or a column formatted as Japanese text) automatically selects the Storage Option of **All in Indexing** for the corresponding field in the FileMaker database file.

Creating an index for any column automatically selects the Storage Option of **Automatically create indexes as needed in Indexing** for the corresponding field in the FileMaker database file.

FileMaker automatically creates indexes as needed. Using `CREATE INDEX` causes the index to be built immediately rather than on demand.

Example

```
CREATE INDEX ON Salespeople.Salesperson_ID
```

DROP INDEX statement

Use the `DROP INDEX` statement to remove an index from a database file. The format of the `DROP INDEX` statement is:

```
DROP INDEX ON table_name.column_name
DROP INDEX ON table_name (column_name)
```

Remove an index when your database file is too large, or you don't often use a field in queries. If your queries are experiencing poor performance, and you're working with an extremely large FileMaker database file with many indexed text fields, consider dropping the indexes from some fields. Also consider dropping the indexes from fields that you rarely use in `SELECT` statements. Dropping an index for any column automatically selects the Storage Option of **None** and clears **Automatically create indexes as needed in Indexing** for the corresponding field in the FileMaker database file.

The `PREVENT INDEX CREATION` attribute is not supported.

Example

```
DROP INDEX ON Salespeople.Salesperson_ID
```

SQL expressions

Use expressions in `WHERE`, `HAVING`, and `ORDER BY` clauses of `SELECT` statements to form detailed and sophisticated database queries. Valid expression elements are:

- Field names
- Constants
- Exponential/scientific notation
- Numeric operators
- Character operators
- Date operators
- Relational operators
- Logical operators
- Functions

Field names

The most common expression is a simple field name, such as `calc` or `Sales_Data.Invoice_ID`.

Constants

Constants are values that do not change. For example, in the expression `PRICE * 1.05`, the value 1.05 is a constant. Or you might assign a value of 30 to the constant `Number_Of_Days_In_June`.

You must enclose character constants in pairs of single quotation marks (`'`). To include a single quotation mark in a character constant enclosed by single quotation marks, use two single quotation marks together (for example, `'Don' 't'`).

For ODBC and JDBC applications, FileMaker accepts the ODBC/JDBC format date, time, and timestamp constants in braces (`{}`), for example:

- `{D '2016-06-05'}`
- `{T '14:35:10'}`
- `{TS '2016-06-05 14:35:10'}`

FileMaker allows the type specifier (`D`, `T`, `TS`) to be in upper case or lower case. You may use any number of spaces after the type specifier, or even omit the space.

FileMaker also accepts SQL-92 syntax ISO date and time formats with no braces:

- `DATE 'YYYY-MM-DD'`
- `TIME 'HH:MM:SS'`
- `TIMESTAMP 'YYYY-MM-DD HH:MM:SS'`

The FileMaker Pro ExecuteSQL function accepts only the SQL-92 syntax ISO date and time formats with no braces.

Constant	Acceptable syntax (examples)
Text	<code>'Paris'</code>
Number	<code>1.05</code>
Date	<code>DATE '2016-06-05'</code> <code>{ D '2016-06-05' }</code> <code>{06/05/2016}</code> <code>{06/05/16}</code> Note The 2-digit year syntax is not supported for the ODBC/JDBC format or the SQL-92 format.
Time	<code>TIME '14:35:10'</code> <code>{ T '14:35:10' }</code> <code>{14:35:10}</code>
Timestamp	<code>TIMESTAMP '2016-06-05 14:35:10'</code> <code>{ TS '2016-06-05 14:35:10' }</code> <code>{06/05/2016 14:35:10}</code> <code>{06/05/16 14:35:10}</code> Make sure Strict data type: 4-Digit Year Date is not selected as a validation option in the FileMaker database file for a field using this 2-digit year syntax. Note The 2-digit year syntax is not supported for the ODBC/JDBC format or the SQL-92 format.

When entering date and time values, match the format of the database file locale. For example, if the database was created on an Italian language system, use Italian date and time formats.

Exponential/scientific notation

Numbers can be expressed using scientific notation.

Example

```
SELECT column1 / 3.4E+7 FROM table1 WHERE calc < 3.4E-6 * column2
```

Numeric operators

You can include the following operators in number expressions: +, -, *, /, and ^ or ** (exponentiation).

You can precede numeric expressions with a unary plus (+) or minus (-).

Character operators

You can concatenate characters.

Examples

In the following examples, `last_name` is 'JONES ' and `first_name` is 'ROBERT ':

Operator	Concatenation	Example	Result
+	Keep trailing blank characters	<code>first_name + last_name</code>	'ROBERT JONES '
-	Move trailing blank characters to the end	<code>first_name - last_name</code>	'ROBERTJONES '

Date operators

You can modify dates.

Examples

In the following examples, `hire_date` is `DATE '2016-01-30'`.

Operator	Effect on date	Example	Result
+	Add a number of days to a date	<code>hire_date + 5</code>	DATE '2016-02-04'
-	Find the number of days between two dates	<code>hire_date - DATE '2016-01-01'</code>	29
	Subtract a number of days from a date	<code>hire_date - 10</code>	DATE '2016-01-20'

Additional examples:

```
SELECT Date_Sold, Date_Sold + 30 AS agg FROM Sales_Data
```

```
SELECT Date_Sold, Date_Sold - 30 AS agg FROM Sales_Data
```

Relational operators

Operator	Meaning
=	Equal
<>	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
LIKE	Matching a pattern
NOT LIKE	Not matching a pattern
IS NULL	Equal to Null
IS NOT NULL	Not equal to Null
BETWEEN	Range of values between a lower and upper bound
IN	A member of a set of specified values or a member of a subquery
NOT IN	Not a member of a set of specified values or a member of a subquery
EXISTS	'True' if a subquery returned at least one record
ANY	Compares a value to each value returned by a subquery (operator must be preceded by =, <>, >, >=, <, or <=); =Any is equivalent to In
ALL	Compares a value to each value returned by a subquery (operator must be preceded by =, <>, >, >=, <, or <=)

Examples

```
SELECT Sales_Data.Invoice_ID FROM Sales_Data
WHERE Sales_Data.Salesperson_ID = 'SP-1'
```

```
SELECT Sales_Data.Amount FROM Sales_Data WHERE Sales_Data.Invoice_ID
<> 125
```

```
SELECT Sales_Data.Amount FROM Sales_Data WHERE Sales_Data.Amount > 3000
```

```
SELECT Sales_Data.Time_Sold FROM Sales_Data
WHERE Sales_Data.Time_Sold < '12:00:00'
```

```
SELECT Sales_Data.Company_Name FROM Sales_Data
WHERE Sales_Data.Company_Name LIKE '%University'
```

```
SELECT Sales_Data.Company_Name FROM Sales_Data
WHERE Sales_Data.Company_Name NOT LIKE '%University'
```

```
SELECT Sales_Data.Amount FROM Sales_Data WHERE Sales_Data.Amount IS NULL
```

```
SELECT Sales_Data.Amount FROM Sales_Data WHERE Sales_Data.Amount IS NOT
NULL
```

```

SELECT Sales_Data.Invoice_ID FROM Sales_Data
  WHERE Sales_Data.Invoice_ID BETWEEN 1 AND 10

SELECT COUNT(Sales_Data.Invoice_ID) AS agg
  FROM Sales_Data WHERE Sales_Data.INVOICE_ID IN (50,250,100)

SELECT COUNT(Sales_Data.Invoice_ID) AS agg
  FROM Sales_Data WHERE Sales_Data.INVOICE_ID NOT IN (50,250,100)

SELECT COUNT(Sales_Data.Invoice_ID) AS agg FROM Sales_Data
  WHERE Sales_Data.INVOICE_ID NOT IN (SELECT Sales_Data.Invoice_ID
  FROM Sales_Data WHERE Sales_Data.Salesperson_ID = 'SP-4')

SELECT *
  FROM Sales_Data WHERE EXISTS (SELECT Sales_Data.Amount
  FROM Sales_Data WHERE Sales_Data.Salesperson_ID IS NOT NULL)

SELECT *
  FROM Sales_Data WHERE Sales_Data.Amount = ANY (SELECT
Sales_Data.Amount
  FROM Sales_Data WHERE Sales_Data.Salesperson_ID = 'SP-1')

SELECT *
  FROM Sales_Data WHERE Sales_Data.Amount = ALL (SELECT
Sales_Data.Amount
  FROM Sales_Data WHERE Sales_Data.Salesperson_ID IS NULL)

```

Logical operators

You can combine two or more conditions. The conditions must be related by AND or OR, such as:

```
salary = 40000 AND exempt = 1
```

The logical NOT operator is used to reverse the meaning, such as:

```
NOT (salary = 40000 AND exempt = 1)
```

Examples

```

SELECT * FROM Sales_Data WHERE Sales_Data.Company_Name
  NOT LIKE '%University' AND Sales_Data.Amount > 3000

SELECT * FROM Sales_Data WHERE (Sales_Data.Company_Name
  LIKE '%University' OR Sales_Data.Amount > 3000)
  AND Sales_Data.Salesperson_ID = 'SP-1'

```

Operator precedence

As expressions become more complex, the order in which the expressions are evaluated becomes important. This table shows the order in which the operators are evaluated. The operators in the first line are evaluated first, and so on. Operators in the same line are evaluated left to right in the expression.

Precedence	Operator
1	Unary '-', Unary '+'
2	^, **
3	*, /
4	+, -
5	=, <>, <, <=, >, >=, Like, Not Like, Is Null, Is Not Null, Between, In, Exists, Any, All
6	Not
7	AND
8	OR

The following example shows the importance of precedence:

```
WHERE salary > 40000 OR hire_date > (DATE '2008-01-30') AND dept =  
'D101'
```

Because AND is evaluated first, this query retrieves employees in department D101 hired after January 30, 2008, as well as every employee making more than \$40,000, no matter what department or hire date.

To force the clause to be evaluated in a different order, use parentheses to enclose the conditions to be evaluated first. For example:

```
WHERE (salary > 40000 OR hire_date > DATE '2008-01-30') AND dept =  
'D101'
```

retrieves employees in department D101 that either make more than \$40,000 or were hired after January 30, 2008.

SQL functions

FileMaker SQL supports many functions you can use in expressions. Some of the functions return characters strings, some return numbers, some return dates, and some return values that depend on conditions met by the function arguments.

Aggregate functions

Aggregate functions return a single value from a set of records. You can use an aggregate function as part of a `SELECT` statement, with a field name (for example, `AVG(SALARY)`), or in combination with a column expression (for example, `AVG(SALARY * 1.07)`).

You can precede the column expression with the `DISTINCT` operator to eliminate duplicate values. For example:

```
COUNT (DISTINCT last_name)
```

In this example, only unique last name values are counted.

Aggregate function	Returns
SUM	The total of the values in a numeric field expression. For example, <code>SUM(SALARY)</code> returns the sum of all salary field values.
AVG	The average of the values in a numeric field expression. For example, <code>AVG(SALARY)</code> returns the average of all salary field values.
COUNT	The number of values in any field expression. For example, <code>COUNT(NAME)</code> returns the number of name values. When using <code>COUNT</code> with a field name, <code>COUNT</code> returns the number of non-null field values. A special example is <code>COUNT(*)</code> , which returns the number of records in the set, including records with null values.
MAX	The maximum value in any field expression. For example, <code>MAX(SALARY)</code> returns the maximum salary field value.
MIN	The minimum value in any field expression. For example, <code>MIN(SALARY)</code> returns the minimum salary field value.

Examples

```
SELECT SUM (Sales_Data.Amount) AS agg FROM Sales_Data
```

```
SELECT AVG (Sales_Data.Amount) AS agg FROM Sales_Data
```

```
SELECT COUNT (Sales_Data.Amount) AS agg FROM Sales_Data
```

```
SELECT MAX (Sales_Data.Amount) AS agg FROM Sales_Data
WHERE Sales_Data.Amount < 3000
```

```
SELECT MIN (Sales_Data.Amount) AS agg FROM Sales_Data
WHERE Sales_Data.Amount > 3000
```

You cannot use an aggregate function as an argument to other functions. If you do, FileMaker returns the error code 8309 (“Expressions involving aggregations are not supported”). For example, the following statement is not valid because the aggregate function `SUM` cannot be used as an argument to the function `ROUND`:

```
SELECT ROUND(SUM(Salary), 0) FROM Payroll
```

However, aggregate functions can use functions that return numbers as arguments. For example, the following statement is valid:

```
SELECT SUM(ROUND(Salary, 0)) FROM Payroll
```

Functions that return character strings

Functions that return character strings

	Description	Example
<code>CHR</code>	Converts an ASCII code to a one-character string	<code>CHR(67)</code> returns C
<code>CURRENT_USER</code>	Returns the login ID specified at connect time	
<code>DAYNAME</code>	Returns the name of the day that corresponds to a specified date	
<code>RTRIM</code>	Removes trailing blanks from a string	<code>RTRIM(' ABC ')</code> returns ' ABC'
<code>TRIM</code>	Removes leading and trailing blanks from a string	<code>TRIM(' ABC ')</code> returns 'ABC'
<code>LTRIM</code>	Removes leading blanks from a string	<code>LTRIM(' ABC')</code> returns 'ABC'
<code>UPPER</code>	Changes each letter of a string to uppercase	<code>UPPER('Allen')</code> returns 'ALLEN'
<code>LOWER</code>	Changes each letter of a string to lowercase	<code>LOWER('Allen')</code> returns 'allen'
<code>LEFT</code>	Returns leftmost characters of a string	<code>LEFT('Mattson',3)</code> returns 'Mat'
<code>MONTHNAME</code>	Returns the names of the calendar month	
<code>RIGHT</code>	Returns rightmost characters of a string	<code>RIGHT('Mattson',4)</code> returns 'tson'
<code>SUBSTR</code> <code>SUBSTRING</code>	Returns a substring of a string, with parameters of the string, the first character to extract, and the number of characters to extract (optional)	<code>SUBSTR('Conrad',2,3)</code> returns 'onr' <code>SUBSTR('Conrad',2)</code> returns 'onrad'
<code>SPACE</code>	Generates a string of blanks	<code>SPACE(5)</code> returns ' '
<code>STRVAL</code>	Converts a value of any type to a character string	<code>STRVAL('Woltman')</code> returns 'Woltman' <code>STRVAL(5 * 3)</code> returns '15' <code>STRVAL(4 = 5)</code> returns 'False' <code>STRVAL(DATE '2008-12-25')</code> returns '2008-12-25'
<code>TIME</code> <code>TIMEVAL</code>	Returns the time of day as a string	At 9:49 PM, <code>TIME()</code> returns 21:49:00
<code>USERNAME</code> <code>USER</code>	Returns the login ID specified at connect time	

Note The `TIME()` function is deprecated. Use the SQL standard `CURRENT_TIME` instead.

Examples

```
SELECT CHR(67) + SPACE(1) + CHR(70) FROM Salespeople
```

```
SELECT RTRIM(' ' + Salespeople.Salesperson_ID) AS agg FROM Salespeople
```

```
SELECT TRIM(SPACE(1) + Salespeople.Salesperson_ID) AS agg FROM
Salespeople
```

```
SELECT LTRIM(' ' + Salespeople.Salesperson_ID) AS agg FROM Salespeople
```

```
SELECT UPPER(Salespeople.Salesperson) AS agg FROM Salespeople
```

```
SELECT LOWER(Salespeople.Salesperson) AS agg FROM Salespeople
```

```
SELECT LEFT(Salespeople.Salesperson, 5) AS agg FROM Salespeople
```

```
SELECT RIGHT(Salespeople.Salesperson, 7) AS agg FROM Salespeople
```

```
SELECT SUBSTR(Salespeople.Salesperson_ID, 2, 2) +
SUBSTR(Salespeople.Salesperson_ID, 4, 2) AS agg FROM Salespeople
```

```
SELECT SUBSTR(Salespeople.Salesperson_ID, 2) +
SUBSTR(Salespeople.Salesperson_ID, 4) AS agg FROM Salespeople
```

```
SELECT SPACE(2) + Salespeople.Salesperson_ID AS Salesperson_ID FROM
Salespeople
```

```
SELECT STRVAL('60506') AS agg FROM Sales_Data WHERE Sales_Data.Invoice
= 1
```

Functions that return numbers

Functions that return numbers	Description	Example
ABS	Returns the absolute value of the numeric expression	
ATAN	Returns the arc tangent of the argument as an angle expressed in radians	
ATAN2	Returns the arc tangent of x and y coordinates as an angle expressed in radians	
CEIL CEILING	Returns the smallest integer value that is greater than or equal to the argument	
DEG DEGREES	Returns the number of degrees of the argument, which is an angle expressed in radians	
DAY	Returns the day part of a date	DAY(DATE '2016-01-30') returns 30
DAYOFWEEK	Returns the day of week (1-7) of a date expression	DAYOFWEEK(DATE '2004-05-01') returns 7
MOD	Divides two numbers and returns the remainder of the division	MOD(10, 3) returns 1

Functions that return numbers	Description	Example
EXP	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument	
FLOOR	Returns the largest integer value that is less than or equal to the argument	
HOUR	Returns the hour part of a value	
INT	Returns the integer part of a number	INT(6.4321) returns 6
LENGTH	Returns the length of a string	LENGTH('ABC') returns 3
MONTH	Returns the month part of a date	MONTH(DATE '2016-01-30') returns 1
LN	Returns the natural logarithm of the argument	
LOG	Returns the common logarithm of the argument	
MAX	Returns the larger of two numbers	MAX(66, 89) returns 89
MIN	Returns the smaller of two numbers	MIN(66, 89) returns 66
MINUTE	Returns the minute part of a value	
NUMVAL	Converts a character string to a number. The function fails if the character string is not a valid number.	NUMVAL('123') returns 123
PI	Returns the constant value of the mathematical constant pi	
RADIANS	Returns the number of radians for an argument that is expressed in degrees	
ROUND	Rounds a number	ROUND(123.456, 0) returns 123 ROUND(123.456, 2) returns 123.46 ROUND(123.456, -2) returns 100
SECOND	Returns the seconds part of a value	
SIGN	An indicator of the sign of the argument: -1 for negative, 0 for 0, and 1 for positive	
SIN	Returns the sine of the argument	
SQRT	Returns the square root of the argument	
TAN	Returns the tangent of the argument	
YEAR	Returns the year part of a date	YEAR(DATE '2016-01-30') returns 2016

Functions that return dates

Functions that return dates	Description	Example
CURDATE CURRENT_DATE	Returns today's date	
CURTIME CURRENT_TIME	Returns the current time	
CURTIMESTAMP CURRENT_TIMESTAMP	Returns the current timestamp value	
TIMESTAMPVAL	Converts a character string to a timestamp	TIMESTAMPVAL('2016-01-30 14:00:00') returns its timestamp value
DATE TODAY	Returns today's date	If today is 11/21/2016, DATE() returns 2016-11-21
DATEVAL	Converts a character string to a date	DATEVAL('2016-01-30') returns 2016-01-30

Note The DATE() function is deprecated. Use the SQL standard CURRENT_DATE instead.

Conditional functions

Conditional functions	Description	Example
CASE WHEN	Simple CASE format	SELECT
	Compares the value of <i>input_exp</i> to the values of <i>value_exp</i> arguments to determine the result.	Invoice_ID, CASE Company_Name WHEN 'Exports UK' THEN 'Exports UK Found' WHEN 'Home Furniture Suppliers' THEN 'Home Furniture Suppliers Found' ELSE 'Neither Exports UK nor Home Furniture Suppliers' END,
	CASE <i>input_exp</i> {WHEN <i>value_exp</i> THEN <i>result...</i> } [ELSE <i>result</i>] END	Salesperson_ID FROM Sales_Data
	Searched CASE format	SELECT
	Returns a result based on whether the condition specified by a WHEN expression is true.	Invoice_ID, Amount, CASE WHEN Amount > 3000 THEN 'Above 3000' WHEN Amount < 1000 THEN 'Below 3000' ELSE 'Between 1000 and 3000' END,
	CASE {WHEN <i>boolean_exp</i> THEN <i>result...</i> } [ELSE <i>result</i>] END	Salesperson_ID FROM Sales_Data

Conditional functions	Description	Example
COALESCE	Returns the first value that is not NULL	<pre>SELECT Salesperson_ID, COALESCE(Sales_Manager, Salesperson) FROM Salespeople</pre>
NULLIF	Compares two values and returns NULL if the two values are equal; otherwise, returns the first value.	<pre>SELECT Invoice_ID, NULLIF(Amount, -1), Salesperson_ID FROM Sales_Data</pre>

FileMaker system objects

FileMaker database files include the following system objects that you can access using SQL queries.

FileMaker system tables

Every FileMaker database file includes two system tables: FileMaker_Tables and FileMaker_Fields. For ODBC applications, these tables are included in the information returned by the catalog function SQLTables. For JDBC applications, these tables are included in the information returned by the DatabaseMetaData method getTables. The tables can also be used in ExecuteSQL functions.

FileMaker_Tables

The FileMaker_Tables table contains information about the database tables defined in the FileMaker file.

The FileMaker_Tables table includes a row for each table occurrence in the relationships graph with the following columns:

- TableName - The name of the table occurrence.
- TableId - The unique ID for the table occurrence.
- BaseTableName - The name of the base table from which the table occurrence was created.
- BaseFileName - The FileMaker filename for the database file that contains the base table.
- ModCount - The total number of times changes to this table's definition have been committed.

Example

```
SELECT TableName FROM FileMaker_Tables WHERE TableName LIKE 'Sales%'
```

FileMaker_Fields table

The FileMaker_Fields table contains information about the fields defined in the FileMaker file.

The FileMaker_Fields table includes the following columns:

- TableName - The name of the table that contains the field.
- FieldName - The name of the field.

- FieldType - The SQL data type of the field.
- FieldId - The unique ID for the field.
- FieldClass - One of three values: Summary, for summary fields; Calculated, for calculated results; or Normal.
- FieldReps - The number of repetitions of the field.
- ModCount - The total number of times changes to this table's definition have been committed.

Example:

```
SELECT * FROM FileMaker_Fields WHERE TableName='Sales'
```

FileMaker system columns

FileMaker adds system columns (fields) to all of the rows (records) in all of the tables that are defined in the FileMaker file. For ODBC applications, these columns are included in the information returned by the catalog function `SQLSpecialColumns`. For JDBC applications, these columns are included in the information returned by the `DatabaseMetaData` method `getVersionColumns`. The columns can also be used in `ExecuteSQL` functions.

ROWID column

The `ROWID` system column contains the unique ID number of the record. This is the same value that the FileMaker Pro `Get(RecordID)` function returns.

ROWMODID column

The `ROWMODID` system column contains the total number of times changes to the current record have been committed. This is the same value that the FileMaker Pro `Get(RecordModificationCount)` function returns.

Example:

```
SELECT ROWID, ROWMODID FROM MyTable WHERE ROWMODID > 3
```

Reserved SQL keywords

This section lists reserved keywords that should not be used as names for columns, tables, aliases, or other user-defined objects. If you are getting syntax errors, these errors may be due to using one of these reserved words. If you want to use one of these keywords, you need to use quotation marks to prevent the word from being treated as a keyword.

For example, the following `CREATE TABLE` statement shows how to use the `DEC` keyword as a data element name.

```
create table t ("dec" numeric)
```

ABSOLUTE	CHAR	CURTIME
ACTION	CHARACTER	CURTIMESTAMP
ADD	CHARACTER_LENGTH	DATE
ALL	CHAR_LENGTH	DATEVAL
ALLOCATE	CHECK	DAY
ALTER	CHR	DAYNAME
AND	CLOSE	DAYOFWEEK
ANY	COALESCE	DEALLOCATE
ARE	COLLATE	DEC
AS	COLLATION	DECIMAL
ASC	COLUMN	DECLARE
ASSERTION	COMMIT	DEFAULT
AT	CONNECT	DEFERRABLE
AUTHORIZATION	CONNECTION	DEFERRED
AVG	CONSTRAINT	DELETE
BEGIN	CONSTRAINTS	DESC
BETWEEN	CONTINUE	DESCRIBE
BINARY	CONVERT	DESCRIPTOR
BIT	CORRESPONDING	DIAGNOSTICS
BIT_LENGTH	COUNT	DISCONNECT
BLOB	CREATE	DISTINCT
BOOLEAN	CROSS	DOMAIN
BOTH	CURDATE	DOUBLE
BY	CURRENT	DROP
CASCADE	CURRENT_DATE	ELSE
CASCADED	CURRENT_TIME	END
CASE	CURRENT_TIMESTAMP	END_EXEC
CAST	CURRENT_USER	ESCAPE
CATALOG	CURSOR	EVERY

EXCEPT	IS	OPTION
EXCEPTION	ISOLATION	OR
EXEC	JOIN	ORDER
EXECUTE	KEY	OUTER
EXISTS	LANGUAGE	OUTPUT
EXTERNAL	LAST	OVERLAPS
EXTRACT	LEADING	PAD
FALSE	LEFT	PART
FETCH	LENGTH	PARTIAL
FIRST	LEVEL	PERCENT
FLOAT	LIKE	POSITION
FOR	LOCAL	PRECISION
FOREIGN	LONGVARBINARY	PREPARE
FOUND	LOWER	PRESERVE
FROM	LTRIM	PRIMARY
FULL	MATCH	PRIOR
GET	MAX	PRIVILEGES
GLOBAL	MIN	PROCEDURE
GO	MINUTE	PUBLIC
GOTO	MODULE	READ
GRANT	MONTH	REAL
GROUP	MONTHNAME	REFERENCES
HAVING	NAMES	RELATIVE
HOUR	NATIONAL	RESTRICT
IDENTITY	NATURAL	REVOKE
IMMEDIATE	NCHAR	RIGHT
IN	NEXT	ROLLBACK
INDEX	NO	ROUND
INDICATOR	NOT	ROW
INITIALLY	NULL	ROWID
INNER	NULLIF	ROWS
INPUT	NUMERIC	RTRIM
INSENSITIVE	NUMVAL	SCHEMA
INSERT	OCTET_LENGTH	SCROLL
INT	OF	SECOND
INTEGER	OFFSET	SECTION
INTERSECT	ON	SELECT
INTERVAL	ONLY	SESSION
INTO	OPEN	SESSION_USER

SET	USER
SIZE	USERNAME
SMALLINT	USING
SOME	VALUE
SPACE	VALUES
SQL	VARBINARY
SQLCODE	VARCHAR
SQLERROR	VARYING
SQLSTATE	VIEW
STRVAL	WHEN
SUBSTRING	WHENEVER
SUM	WHERE
SYSTEM_USER	WITH
TABLE	WORK
TEMPORARY	WRITE
THEN	YEAR
TIES	ZONE
TIME	
TIMESTAMP	
TIMESTAMPVAL	
TIMEVAL	
TIMEZONE_HOUR	
TIMEZONE_MINUTE	
TO	
TODAY	
TRAILING	
TRANSACTION	
TRANSLATE	
TRANSLATION	
TRIM	
TRUE	
TRUNCATE	
UNION	
UNIQUE	
UNKNOWN	
UPDATE	
UPPER	
USAGE	

Index

A

ABS function 31
aggregate functions in SQL 29
ALL operator 26
ALTER TABLE (SQL statement) 22
AND operator 27
ANY operator 26
ATAN function 31
ATAN2 function 31

B

BaseFileName 34
BaseTableName 34
BETWEEN operator 26
binary data, use in SELECT 15
blank characters 25
blank value in columns 18
BLOB data type, use in SELECT 15

C

CASE WHEN function 33
CAST function 16
CEIL function 31
CEILING function 31
character operators in SQL expressions 25
CHR function 30
COALESCE function 34
column aliases 8
constants in SQL expressions 24
container field
 stored externally 21
 with CREATE TABLE statement 21
 with GetAs function 16
 with INSERT statement 18
 with PutAs function 18
 with SELECT statement 16
 with UPDATE statement 19
CREATE INDEX (SQL statement) 22
CREATE TABLE (SQL statement) 20
CURDATE function 33
CURRENT USER function 30
CURRENT_DATE function 33
CURRENT_TIME function 33
CURRENT_TIMESTAMP function 33
CURRENT_USER function 30
cursors in ODBC 14
CURTIME function 33
CURTIMESTAMP function 33

D

date formats 24
DATE function 33
date operators in SQL expressions 25
DATEVAL function 33
DAY function 31
DAYNAME function 30
DAYOFWEEK function 31
DEFAULT (SQL clause) 20
DEG function 31
DEGREES function 31
DELETE (SQL statement) 17
DISTINCT operator 8
documentation 5
DROP INDEX (SQL statement) 23

E

empty string, use in SELECT 15
ExecuteSQL function 6, 7
EXISTS operator 26
EXP function 32
exponential notation in SQL expressions 25
expressions in SQL 23
EXTERNAL (SQL clause) 21

F

FETCH FIRST (SQL clause) 13
field names in SQL expressions 23
field repetitions 17, 20
FieldClass 35
FieldId 35
FieldName 34
FieldReps 35
FieldType 35
FileMaker Server documentation 5
FileMaker_Fields 34
FileMaker_Tables 34
files, use in container fields 16
FLOOR function 32
FOR UPDATE (SQL clause) 14
FROM (SQL clause) 9
FULL OUTER JOIN 10
functions in SQL expressions 29

G

GetAs function 16
GROUP BY (SQL clause) 11

H

HAVING (SQL clause) 11
 HOUR function 32

I

IN operator 26
 INNER JOIN 10
 INSERT (SQL statement) 17
 INT function 32
 IS NOT NULL operator 26
 IS NULL operator 26

J

JDBC client driver
 portals 7
 Unicode support 7
 join 10

K

keywords, reserved SQL 36

L

LEFT function 30
 LEFT OUTER JOIN 10
 LENGTH function 32
 LIKE operator 26
 LN function 32
 LOG function 32
 logical operators in SQL expressions 27
 LOWER function 30
 LTRIM function 30

M

MAX function 32
 MIN function 32
 MINUTE function 32
 MOD function 31
 ModCount 34, 35
 MONTH function 32
 MONTHNAME function 30

N

NOT IN operator 26
 NOT LIKE operator 26
 NOT NULL (SQL clause) 20
 NOT operator 27
 null value 18
 NULLIF function 34
 numeric operators in SQL expressions 25
 NUMVAL function 32

O

ODBC client driver
 portals 7
 Unicode support 7
 ODBC standards compliance 7
 OFFSET (SQL clause) 13
 online documentation 5
 operator precedence in SQL expressions 28
 OR operator 27
 ORDER BY (SQL clause) 12
 OUTER JOIN 10

P

PDFs 5
 peer rows 13, 14
 PI function 32
 portals 7
 positioned updates and deletes 14
 PREVENT INDEX CREATION 23
 PutAs function 18, 19

R

RADIANS function 32
 relational operators in SQL expressions 26
 reserved SQL keywords 36
 RIGHT function 30
 RIGHT OUTER JOIN 10
 ROUND function 32
 ROWID system column 35
 ROWMODID system column 35
 RTRIM function 30

S

scientific notation in SQL expressions 25
 SECOND function 32
 SELECT (SQL statement) 8
 binary data 15
 BLOB data type 15
 empty string 15
 SIGN function 32
 SIN function 32
 sort order 12
 SPACE function 30
 SQL aggregate functions 29
 SQL expressions 23
 character operators 25
 constants 24
 date operators 25
 exponential or scientific notation 25
 field names 23
 functions 29
 logical operators 27
 numeric operators 25
 operator precedence 28
 relational operators 26

SQL standards compliance 7
 SQL statements
 ALTER TABLE 22
 CREATE INDEX 22
 CREATE TABLE 20
 DELETE 17
 DROP INDEX 23
 INSERT 17
 reserved keywords 36
 SELECT 8
 supported by client drivers 7
 TRUNCATE TABLE 21
 UPDATE 19
 SQL_C_WCHAR data type 7
 SQL-92 7
 SQRT function 32
 standards compliance 7
 string functions 30
 STRVAL function 30
 subqueries 18
 SUBSTR function 30
 SUBSTRING function 30
 syntax errors 36
 system tables 34

T

table aliases 8, 9
 TableId 34
 TableName 34
 TAN function 32
 time formats 24
 TIME function 30
 timestamp formats 24
 TIMESTAMPVAL function 33
 TIMEVAL function 30
 TODAY function 33
 TRIM function 30
 TRUNCATE TABLE (SQL statement) 21

U

Unicode support 7
 UNION (SQL operator) 12
 UNIQUE (SQL clause) 20
 UPDATE (SQL statement) 19
 UPPER function 30
 USERNAME function 30

V

VALUES (SQL clause) 17

W

WHERE (SQL clause) 10
 WITH TIES (SQL clause) 13

Y

YEAR function 32