The Missing FileMaker™ 12 ExecuteSQL Reference

19 OCT 2012

Beverly Voth

There seem to be many questions about the usage of SQL (Structured Query Language) with the **ExecuteSQL function in FileMaker 12**. This tutorial attempts to explain some of the SQL terms, if you are new to writing SQL statements. Since there are already many examples of how to write the ExecuteSQL queries, links to these will be listed at the end of this article. If you don't need to learn the terms, jump right to the Helpful Example Databases section, below. There you will find links to solutions that help you create and test your queries.

This is not a complete SQL guide, as other databases may use other syntax. This is not a complete FileMaker and SQL guide, as FileMaker may be an ODBC source and the SQL queries made against it may vary from the terms used by ExecuteSQL(). This is not a complete FileMaker and ESS guide using SQL calls (if using Import or Execute SQL script steps or ExecuteSQL() function or ESS). It may not have all the nuances needed for other data sources. This is the ExecuteSQL() function reference for which you've been waiting.

The FileMaker 12 ODBC and JDBC Guide is helpful, but it has uses outside (and beyond) the ExecuteSQL() function. Any discrepancies between the reference and the function will be noted here, if possible.

To become familiar with the function, start with the ExecuteSQL, FM12 help topic. The "?" arguments are used with the ExecuteSQL(sqlQuery; fieldSeparator; rowSeparator {;arguments...}) function to pass parameters to the query. The "?" can be used in any part of the SELECT statement, although typically is used to pass search criteria in the WHERE clause.

FMP12 ExecuteSQL() is for SELECT Only

At this time, using the FileMaker 12 ExecuteSQL function, the SQL statement **SELECT** is a way to return delimited results to a field or variable (with these optional features):

- find (with or without comparison criteria)
- use constants and literals (as results and as comparison criteria)
- concatenate, calculate & summarize data (for results)
- sort (by multiple fields/columns of results)
- join (create temporary relationships, including self-joins for results)
- union (stack or concatenate SELECTs from several tables and show in results)
- group results

The xDBC Guide, Page 37, Supported Standards (with additional notes):
These terms are defined in this article:

```
SELECT [DISTINCT] {* | column_expression [[AS] column_alias],...}
    use fields, constants, calculations and functions
FROM table_name [[AS] table_alias], ...
    [ JOIN table_name ON matches ]
    list of tables or explicit relationships
[ WHERE expr1 rel_operator expr2 ]
    comparisons with AND &amp;amp; OR
    comparisons using LIKE, IN, or BETWEEN..AND
    can contain nested SELECT for IN
[ GROUP BY {column_expression, ...} ]
    list all fields NOT in an aggregate function
[ HAVING expr1 rel_operator expr2 ]
    comparisons using aggregate functions
[ UNION [ALL] (SELECT...) ]
    each SELECT must return the same # of columns
[ ORDER BY {sort_expression [DESC | ASC]}, ... ]
    comma-separated sort list
```

Find Everything in a Table

In its most basic statement, SQL SELECT will find all columns (fields) from a single table:

```
SELECT * FROM mytable
```

You must have something to find (SELECT *) and a table (FROM <<mytable>>). You must use FileMaker table occurrences (T.O.) as named on the Relationship Graph for your ExecuteSQL queries, but they will evaluate as if the base table had been selected. The function will not "filter" from the relationship (or any) context. The table occurrence must be ON the relationship graph in the file where the ExecuteSQL is performed, as you cannot query an unknown datasource. And the table must have at least one record to return any result (see the system functions in the example database).

> NOTE: there is no differentiation between FileMaker table occurrences and External SQL Sources (ESS) table occurrences on the Relationship graph. Some of the SQL functions may or may not work as expected.

No FileMaker layouts or relationships are required to make ExecuteSQL return results. Keep in mind that if you are using fields to supply the criteria (any clause), that you may need to be on a layout that shows the field available in the Specify Calculation dialog. The "found sets of records" are not used when using the ExecuteSQL function. All records are available (with permission) to make the queries. "Field formatting" is also ignored.

The "*" (asterisk) is a shortcut for "all fields/columns" and since we have not used a WHERE clause, all records and all fields will be returned from the specified table occurrence.

```
ExecuteSQL ( " SELECT * FROM mytable "
   ; "" ; "" ) // use the default delimiters
```

If you have named your table occurrence with characters (spaces, reserved words or other characters that may return an error to your result), then you may quote it:

```
ExecuteSQL ( " SELECT * FROM \"my table\" "
   ; "" ; "" )
```

> Names are case insensitive for **SQL tables and columns**. "My Table" is the same as "my table". The SQL statement commands are also not case sensitive, so that " SELECT * FROM " is the same as " select * from ". My preference is to use uppercase for the keywords, so they are easy to find in my scripts and calculations.

Find Specific Fields in a Table

Add the names of fields to return just those results. Since no layout is used, any fields in the table occurrence can be used. Quote the field names if they contain spaces or other characters that might return an error. Field names that use reserved words must be quoted, too:

```
ExecuteSQL ( " SELECT FirstName, LastName, State, Zipcode
   FROM myContacts "
   ; "" ; "" )

ExecuteSQL ( " SELECT \"First Name\", \"Last Name\",
   State, Zipcode
   FROM myContacts "
   ; "" ; "" ) // spaces in field names

ExecuteSQL ( " SELECT \"date\", amount
   FROM sales
WHERE \"date\" >= '2012-01-01'
AND amount > 500 "
   ; "" ; "" ) // date is a RESERVED WORD
```

See the references for ways to prevent field and table name changes from breaking your SQL queries. The "Robust Coding" blogs and the custom functions may be used for ideas on how to make your queries "dynamic".

ORDER BY = Sorting with ExecuteSQL!

What a nice list of contacts we have. Can we sort them? Yes, the SQL clause ORDER BY is our sorting mechanism. It uses a comma-delimited list of fields/columns to sort. You can optionally specify DESC (descending sort order). The ASC (ascending sort order) is the default so it is un-necessary to specify. Remember that ORDER BY is always the last clause in your SQL statement.

The following will sort the last name field in a reverse order and sort the first name field (ASC by default). In the FileMaker Sort dialog, you make the same kind of sorting order.

```
ExecuteSQL ( " SELECT FirstName, LastName, State, Zip
    FROM myContacts
    ORDER BY LastName DESC, FirstName "
    ; "" ; "" ) // ORDER BY 2 DESC, 1 ... (coumn number) is a valid method, too!
```

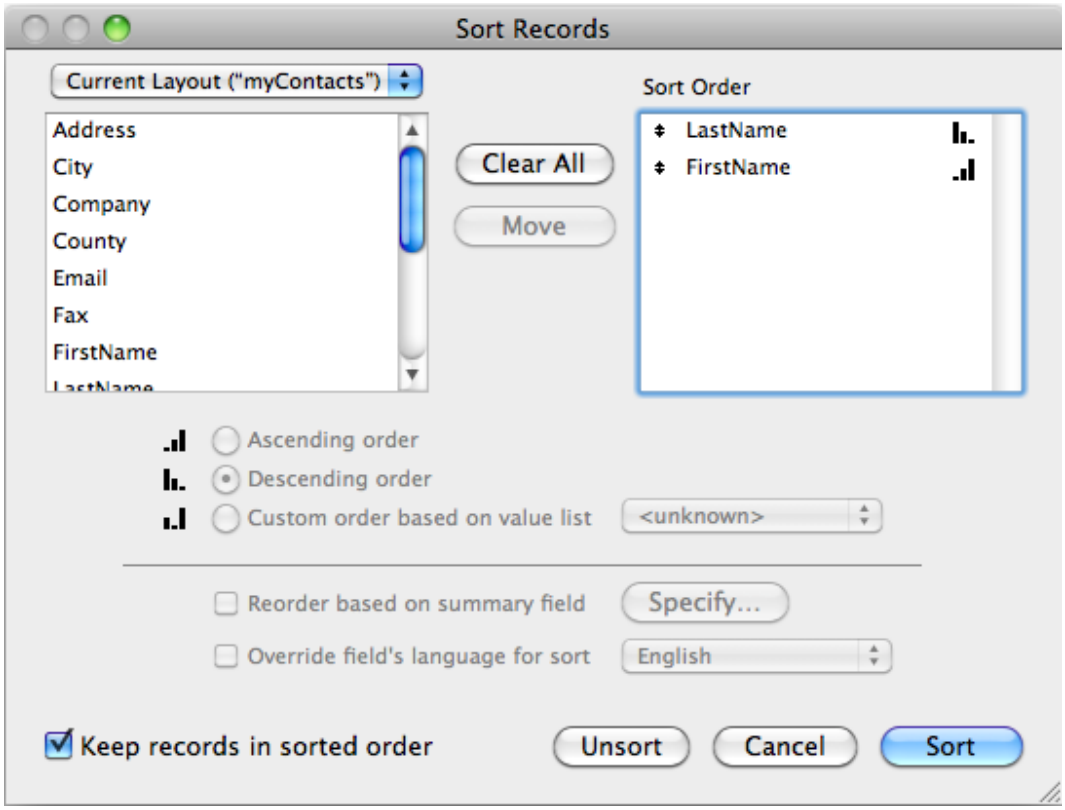

Figure 1 Sort Record dialog

Concatenation & Calculations in the SELECT Statement

It's often convenient to return the result combined in a way that is different than just 'field, comma/tab, field'. Concatenation is a way to make a query on several fields pushing them into one "column". The character "+" or "||" can be used to concatenate in a query when used in ExecuteSQL. I found that "||" (the double pipe) worked every time, the "+" was a little more particular.

```
/* concatenate with "+" test */

Let (
   [ $query = " SELECT LastName+', '+FirstName
      FROM myContacts
      WHERE LastName LIKE ?
         AND FirstName LIKE ? "

   ; $result = ExecuteSQL ( $query
      ; "" ; ""
      ; "Ab%" ; "A%" )

   ]; $result
) // SUCCESS: Abanour, Alyce
```

```
/* concatenate, place returns between fields */

Let (
   [ $query = " SELECT
      FirstName + ' ' + LastName
      , Address
      , City + ', ' + State + ' ' + Zip
      , '---'
      FROM myContacts
      WHERE LastName LIKE ?
         AND FirstName LIKE ?
      ORDER BY state, zip "

   ; $result = ExecuteSQL ( $query
      ; Char(13) ; Char(13)
      ; "Ab%" ; "A%" )
```

```
   ]; $result
) // SUCCESS - ready to send to labels

/*
Alden Abee
1301 Century Cir
Wasilla, AK 99654
---
Asa Abriola
Garden Oaks Shopping
Camden, AR 71701
---
Ambrose Abrego
220 W 5th St
Hope, AR 71801
---
*/
```

```
/* concatenate with "||" test */

Let (
   [ $query = " SELECT LastName||',    '||FirstName
      FROM myContacts
      WHERE LastName LIKE ?
         AND FirstName LIKE ? "

   ; $result = ExecuteSQL ( $query
      ; "" ; ""
      ; "Ab%" ; "A%" )

   ]; $result
) // SUCCESS: Abanour,    Alyce
```

Calculations can be performed on the columns and results returned to a new column with a named alias.

```
/* calculate qty x price - note: 4 columns will be returned */

Let (
    [ $query = " SELECT
        productID
        , qty
        , price
        , qty*price AS extendedPrice
        FROM lineItems
        WHERE orderID = ? "

    ; $result = ExecuteSQL ( $query
        ; char(9) ; ""
        ; orders::orderID )

    ]; $result
) // no example in sample database
```

SQL Operators

The following *Operators* are valid in ExecuteSQL(). Mathematical errors (division by zero, for example) are not valid in ExecuteSQL(). And, of course, these operators are meant to work on fields/columns that return numeric values.

+	Addition (do not confuse it with Concatenation of strings)
-	Subtraction
*	Multiplication
/	Division
%	Modulus (do not confuse it with the LIKE wildcard)
()	Use parenthesis for clarity

Table 1 Operators

Adding Literals

The spacing in the Concatenation tests above uses literals. Any text can be in single quotes. Numbers do not need to be quoted. Constants are another form of Literal that might be used in the WHERE clause (search criteria). SQL functions can also be included in the query.

```
/* literal test */

Let (
   [ $query = "SELECT 'ABC123-' || zip || ' ' AS lit_text,
      123, CURRENT_DATE AS cur_date
      FROM myContacts
      WHERE State = ?
      ORDER BY zip DESC "

   ; $result = ExecuteSQL ( $query
      ; Char(9) ; ""
      ; "WA" )

   ]; $result
) // SUCCESS
```

Using SQL Functions

System Functions

SQL has some functions that maybe used in the queries. Each SQL db may have a different set of functions. The last query contains the *System* function "**CURRENT_DATE**" and returns the date in the "YYYY-MM-DD" format. Other System functions that work within ExecuteSQL:

- **CURRENT_TIME** = "hh:mm:ss" (24-hour time)
- **CURRENT_TIMESTAMP** = "YYYY-MM-DD hh:mm:ss" (24-hour time)
- **CURRENT_USER** = returns same as the Get (AccountName) function in FileMaker (also **USER** & **USERNAME** return the same value)

DateTime Functions

Date & Time functions that work well in ExecuteSQL() and may be used with the System functions, above or your date fields or even the properly formatted Date/Time text (in single quotes 'YYYY-MM-DD hh:mm:ss'):

COALESCE(*date,"*)	Date +/- (add., subr.) days	DATE()
CURDATE	CURTIME	CURTIMESTAMP
MONTHNAME()	DAYNAME()	DAYOFWEEK()
MONTH()	DAY()	YEAR()
HOUR()	MINUTE()	SECOND()
STRVAL()	TIME()	TODAY
	EXTRACT(*part FROM..)*	

Table 2 DateTime Functions

```
/*
Removes duplicates based on listed fields.
And extract just a Year part from a date field
*/

Let (
  [ $query = " SELECT DISTINCT YEAR(s.\"date\")
     FROM sales_related_sales AS s
     WHERE s.amount > ?
     ORDER BY  s.amount DESC "

  ; $result = ExecuteSQL ( $query
     ; "" ; "" ; 450 )

  ]; $result
)
```

DAYNAME() returns "Monday", "Tuesday", etc. and DAYOFWEEK() return a number (Sunday = 1). Formatting dates may be done with COALESCE(), STRVAL() and NUMVAL() conversion functions. Dates may also have addition and subtraction performed on them (+/- days only).

Conversion Functions

CAST() is a SQL function to change the datatype of a field. Number to text, for example. CAST(m.number AS VARCHAR). See the xDBC Guide (Chapter 8) for "Mapping FileMaker fields to ODBC data types". CAST in combination with other functions may not return results as expected, so use cautiously.

text	*number*	*date*	*time*	*timestamp*
VARCHAR	DOUBLE	DATE	TIME	TIMESTAMP

Table 3 CAST Type Mappings

COALESCE(), *STRVAL()*, *NUMVAL()* may all be considered "conversion" functions, as they return different values when used around certain other functions. COALESCE for dates, times and timestamps returns mm/dd/yyyy h:mm:ss A instead of yyyy-mm-dd hh:mm:ss formatting. Remember to add the comma+double-single-quote (,'') as part of the function.

STRVAL(CURRENT_TIMESTAMP) seems to return the same results as COALESCE() with date fields and SQL datetime functions. STRVAL on numbers returns text, but since the ExecuteSQL() returns text this seems redundant.

NUMVAL() will convert dates to number of days similar to the FileMaker function GetAsNumber(Get(CurrentDate)). Time is converted by NUMVAL() to the number of seconds similar to the FileMaker function GetAsNumber(Get(CurrentTime)). NUMVAL('num_as_string') returns the number.

String Functions

String functions, such as LOWER & UPPER are most useful when needed to change the case of a field to test with the LIKE comparison in the WHERE clause. Since the comparison is case sensitive, a search for " LIKE 'A%' " will only return the column/field that begins with the capital letter "A". Any field that begins with "a" will not be in the result. The string functions can also be used in the SELECT to change the case of the result:

```
/* test upper and lower */

Let (
   [ $query = " SELECT LOWER(FirstName), LastName
      FROM myContacts
      WHERE UPPER(LastName) LIKE ? "

   ; $result = ExecuteSQL ( $query
      ; Char(9) ; ""
      ; "AA%" )

   ]; $result
) // SUCCESS - christen    Aalund
```

```
/* LOWER() test */

Let (
   [ $query = " SELECT LOWER(lastname+', '+firstname)
      FROM mycontacts
      WHERE lastname LIKE ?
         AND firstname LIKE ? "

   ; $result = ExecuteSQL ( $query
      ; "" ; ""
      ; "Ab%" ; "A%" )

   ]; $result
) // SUCCESS: abbott, ashley
```

There are many SQL string functions that have not been tested for this article. Some string functions have been tested, but do not work as expected. The SQL string functions that seem to work well with Execute SQL:

CHR()	COALESCE()	LEFT()	LENGTH()	LOWER()
LTRIM()	MID()	RIGHT()	RTRIM()	SPACE()
SUBSTR()	SUBSTRING()	TRIM()	UPPER()	

Table 4 String Functions

```
/* SUBSTRING() test.
this is similar to FMP Middle() function */

Let (
   [ $query = " SELECT SUBSTRING(lastname,1,4), firstname
      FROM mycontacts
      WHERE lastname LIKE ?
         AND firstname LIKE ? "

   ; $result = ExecuteSQL ( $query
      ; "" ; ""
      ; "Ab%" ; "A%" )

   ]; $result
) // SUCCESS
```

COALESCE() is a special string function to remove (trim) concatenated strings that might otherwise return NULL and the concatenated text. The COALESCE() function returns a copy of its first non-NULL argument, or NULL if all arguments are NULL. Coalesce() must have at least 2 arguments. If LastName is NULL (empty), you'll get (, fname), using COALESCE to "trim":

```
COALESCE( Lastname || ',  ', '' ) || FirstName
```

Aggregate Functions

Aggregate (summary) SQL function work with ExecuteSQL and return the same results as the summary fields in FileMaker. If fields (other than used in aggregate functions) are to be returned in the result, the GROUP BY clause is used in the query to name the other fields.

AVG()	Computes the average value of a column
COUNT()	Counts the rows defined by the non-NULL fields
COUNT(*)	Counts ALL rows by defined fields
MIN()	Finds the minimum value in a column
MAX()	Finds the maximum value in a column
SUM()	Computes the sum of column values
STDEV()	Statistical Standard Deviation of all values
STDEVP()	Statistical Standard Deviation for the Population for all values

Table 5 SQL Aggregate Functions

```
/*
SQL SUM function - if other fields are requested,
they must be listed in a GROUP BY clause
Aggregates, are also be used in the HAVING clause,
if comparing summary results.
SUM returns the Total amount &amp;amp; an alias is assigned to the column,
so that it can be used in the sort.
*/

Let (
    [ $query = " SELECT '#'||s.salespersonID
        , SUM( s.amount ) AS sum_amount
        FROM sales_related_sales AS s
        GROUP BY s.salespersonID
        ORDER BY sum_amount DESC "

    ; $result = ExecuteSQL( $query ; ",  "; "" )

    ]; $result
) // SUCCESS
```

Math Functions

Some SQL *Math functions* that may or may not work with ExecuteSQL (tested for this article):

ABS()	√	ASIN()	X	ATAN()	√	ATN2()	X
ATAN2()	√	CEIL()	√	CEILING()	√	COS()	√
COT()	X	DEG()	√	DEGREES()	√	EXP()	√
FLOOR()	√	LOG()	√	LOG2()	X	LOG10()	X
MOD()	√	PI()	√	POWER()	X	RADIANS()	√
RAND()	X	ROUND()	√	SIGN()	√	SIN()	√
SQRT()	√	SQUARE()	X	TAN()	√	TRUNCATE()	X

Table 6 SQL Math Functions

```
/* SQL function ROUND(number,decimal_places)
does not return an error, but does not alter format
of number in the result.
*/

Let (
    [ $query = " SELECT s.\"date\", ROUND(s.amount,2) AS amt_fmt
        FROM sales_related_sales AS s
        WHERE s.salespersonID = ?
            AND s.amount >= ?
        ORDER BY s.amount DESC "

    ; $header = "date" &amp;amp; char(9) &amp;amp; "amount¶"

    ; $result = TextStyleAdd($header;Bold) &amp;amp;
        ExecuteSQL ( $query ; char(9) ; ""
        ; salesperson::salespersonID ; 100 )

    ]; $result
)// does NOT round to two decimals, but does not fail
```

SQL Logical Functions

The CASE SQL function, much like the FileMaker Case() function, allows multiple tests and results & includes an optional default. The SQL CASE function for use in ExecuteSQL has two variations:

1. Simple CASE expression - this uses the expression as in input and the values are used in the WHEN 'test'

```
CASE input
   WHEN value1 THEN result1
   { WHEN value2 THEN result2 }
   { ... }
   { ELSE result3 }
END
```

2. Searched CASE expression - each 'test' expression can be different.

```
CASE
   WHEN expr1 THEN result1
   { WHEN expr2 THEN result2 }
   { ... }
   { ELSE result3 }
END
```

There are two other logical functions found in some SQL systems: IIF [or IF] and CHOOSE. None of these worked with ExecuteSQL(), so the suggestion to use the CASE is presented in the example file.

```
/*
Since CASE works and IIF or IF don't...
*/

Let (
  [ $query = " SELECT p.name,
      CASE
         WHEN p.salespersonID > 3
         THEN 'x'
```

```
          ELSE 'o'
       END, p.salespersonID
       FROM salesperson AS p "

  ; $result = ExecuteSQL ( $query ; ", " ; "" )

  ]; $result
) // SUCCESS
```

```
/*
CHOOSE ( index, val_1, val_2 [, val_n ] )
- Returns the item at the specified index from a list of values.
does not work, so changed to CASE
*/

Let (
  [ $query = " SELECT s.year_month, s.salespersonID, s.amount,
     CASE MONTH(s.\"date\")
        WHEN 1 THEN 'JAN'
        WHEN 2 THEN 'FEB'
        WHEN 3 THEN 'MAR'
        WHEN 4 THEN 'APR'
        WHEN 5 THEN 'MAY'
        WHEN 6 THEN 'JUN'
        WHEN 7 THEN 'JUL'
        WHEN 8 THEN 'AUG'
        WHEN 9 THEN 'SEP'
        WHEN 10 THEN 'OCT'
        WHEN 11 THEN 'NOV'
        WHEN 12 THEN 'DEC'
        END
     FROM sales_related_sales AS s
     WHERE LEFT(s.year_month,4) = '2010'
        AND s.salespersonID = 1 "

  ; $result = ExecuteSQL ( $query ; ", " ; "" )

  ]; $result
) // since the CHOOSE() function doesn't work,
// this was revised to use the CASE function
```

ExecuteSQL Meta functions

These functions are available for use with ExecuteSQL and query the database:

FUNCTION:	*Columns returned:*
FILEMAKER_TABLES	1. TableName 2. TableID 3. BaseTableName 4. BaseFileName 5. ModCount
FILEMAKER_FIELDS	1. TableName 2. FieldName 3. FieldType (the SQL data type, not the FileMaker data type) 4. FieldID 5. FieldClass (Normal, Summary, Calculated) 6. FieldReps 7. ModCount

Table 7 ExecuteSQL META Functions

```
Let (
  [ $query = " SELECT *
    FROM FileMaker_Tables "

  ; $header = "TableName, TableID, BaseTableName, BaseFileName, ModCount¶"

  ; $result = $header & ExecuteSQL ( $query
    ; ", " ; "" )

  ]; $result

) //
```

```
Let (
  [ $query = " SELECT *
    FROM FILEMAKER_FIELDS "

  ; $result = ExecuteSQL ( $query
    ; ", " ; "" )
```

```
   ]; $result
) //
```

Find Criteria is in the WHERE Clause

For find criteria, the **values** are case sensitive or they will not match in SQL. Changing the field Options, Storage, Indexing, Default language settings may not help. "A" is not the same as "a" when making SQL queries.

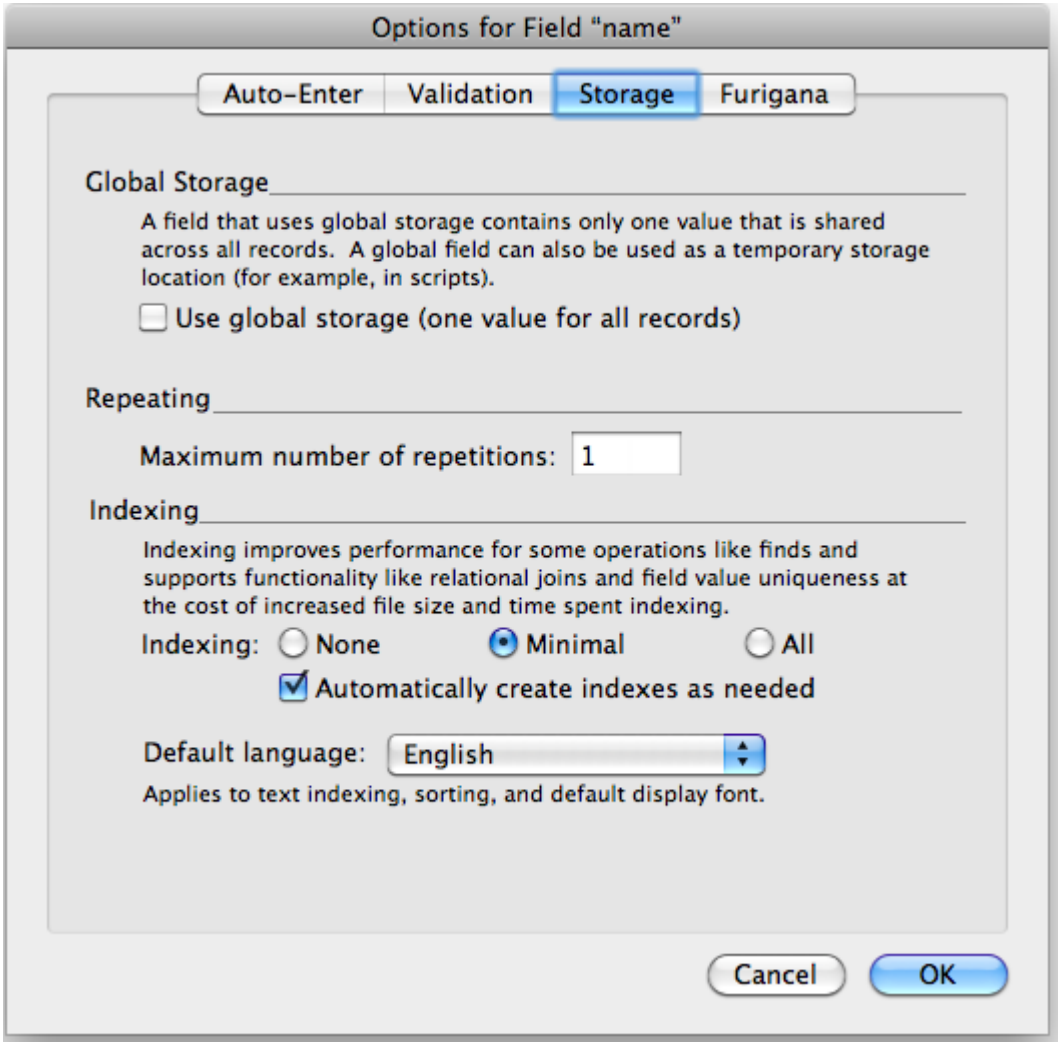

Figure 2 Field Options Indexing dialog

Comparison Operators

Comparison Operators are used in the WHERE clauses. Just as these are used in the FileMaker Relationship Graph when connecting two or more fields, operators make straight comparisons between fields, fields and variables, or fields and constants:

=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to

Special Comparison Operators

SQL has some special comparison operators:

BETWEEN	(with AND) is for inclusive ranges
LIKE	(with wildcards) is used for searches that find parts of words or numbers
IN	(comma delimited) is used to search for a list of values
NOT	another term that may be used to "omit" from your result set
NULL	constant is often used to find, but only if the field/column is not empty
AND	Boolean true if all of the expressions are true
OR	Boolean true if any of the expression are true
ANY	Used with subqueries - returns TRUE when the comparison specified is TRUE for any pair (scalar_expression, x) where x is a value in the single-column set; otherwise, returns FALSE.
ALL	Used with subqueries - returns TRUE when the comparison specified is TRUE for all pairs (scalar_expression, x), when x is a value in the single-column set; otherwise returns FALSE.
EXISTS	Specifies a subquery to test for the existence of rows.

The AND & OR operators may be used (in any combination) to narrow down your choices. If making more than one "request" a single WHERE clause uses multiple **AND** & OR **operators**. These requests can be nested and inserted in parenthesis to make the intent clear.

```
WHERE ( x = 1 OR y = 2 ) // either can be true
  AND ( z = 3 ) // and this must be true
```

IS	means the same as "="
IS NULL	tests for no value
!	sometimes is used as a shortcut to NOT
!=	same as "<>"

Table 8 Alternate Comparison Operators

```
  WHERE s.sales_date IS NOT NULL
or
  WHERE s.sales_date != NULL
```

```
Let (
   [ $query = " SELECT s.salesPersonID, s.\"date\", s.amount
       FROM sales_related_sales AS s
       WHERE s.\"date\" BETWEEN ? AND ?
       ORDER BY s.salesPersonID DESC, s.\"date\" "

   ; $header = "Sales Between 2010-01-01 and " &
        Date(7; 0 ; 2010 ) & "¶"
   ; $header = $header & "ID" &amp;amp; char(9) & "sales¶"

   ; $result = ExecuteSQL ( $query ; char(9) ; ""
       ; "2010-01-01"
       ; Date(7; 0 ; 2010 ) )

   ]; TextStyleAdd($header;Bold) & $result
) // the dates are supplied as "literal" (yyyy-mm-dd)
 // and with FileMaker Date functions
```

LIKE with Wildcards

The LIKE keyword in a WHERE clause gives us the ability to use some wildcard characters similar to the use of symbols in FileMaker find requests. The two symbols (characters) that work with ExecuteSQL are the "%" (percent character, meaning one or more) and "_" (underscore, meaning one character). These can be used anywhere within the string to be compared. The "%" wildcard was used in several other examples in this article.

```
/* using Wildcard "_" in LIKE */
Let (
   [ $query = " SELECT firstname, lastname FROM mycontacts
       WHERE lastname LIKE ?
       ORDER BY lastname "

   ; $result = ExecuteSQL ( $query
       ; "" ; ""
       ; "A_a%" ) // test for uppercase A
      // followed by any character
      // followed by lowercase a
   ]; $result
) // SUCCESS
```

WHERE ... IN ({ SELECT ... })

The IN keyword used for the WHERE clause, takes a comma-delimited list and searches by each of the values (an OR search). If the list is composed of numbers, the list is just comma-delimited. If the list is TEXT, then the values, must be enclosed with single quotes ('abc','def','ghi','jkl'). These values will be automatically be quoted as needed, if you nest another SELECT inside. Only one column/field should be returned in the nested SELECT results.

```
/* WHERE ... IN not dynamic */

Let (
  [ $query = " SELECT firstname,lastname,city,state
```

```
      FROM myContacts
      WHERE state IN ('MI','IN','OH')
        AND LOWER(lastname) LIKE 'z%'
        AND LOWER(firstname) LIKE 'a%'
      ORDER BY state, city, lastname "

   ; $result = ExecuteSQL ( $query
      ; "," ; ""
      )

   ]; $result
) // SUCCESS
```

```
/*
SELECT WHERE ... IN () with ? arguments
*/

Let (
   [ $query = " SELECT firstname,lastname,city,state
      FROM myContacts
      WHERE state IN (?, ?, ?)
        AND LOWER(lastname) LIKE 'z%'
        AND LOWER(firstname) LIKE 'a%'
      ORDER BY state, city, lastname "

   ; $result = ExecuteSQL ( $query
      ; "," ; ""
      ; "MI" ; "IN" ; "OH" )

   ]; $result
) // same query as above, with the arguments
// as multiple inside the IN
```

Subqueries

An example of nested SELECTs uses two tables (unrelated, but having a common key). This could be done with a JOIN, but is just a demonstration of finding one column for use in the WHERE...IN. This example finds all the distinct salespersonID in the sales records. It narrows down the list by finding those who had sales in the year 2009. That "list" gets put into the IN keyword for use with the outer SELECT. The results listed those

salespersons who had sales in 2009.

```
/*
nesting SELECTS for WHERE clause
*/

Let (
  [ $query = " SELECT
    s.name
    FROM salesperson AS s
    WHERE s.salespersonID IN (
      SELECT DISTINCT
        sales.salespersonID
      FROM sales_related_sales AS sales
      WHERE YEAR(sales.\"date\") = ?
      )
    ORDER BY s.name "

  ; $result = ExecuteSQL ( $query
    ; "," ; ""
    ; 2009 )

  ]; $result
) //
```

The WHERE comparisons of ALL, ANY (or SOME) and EXISTS may be used to subquery and narrow down a selection.

SELECT DISTINCT

Return unique values by using the DISTINCT keyword with SELECT. Some of the SQL functions may also use the DISTINCT keyword, but have been untested for this article. Example uses of DISTINCT are found here and in other articles. If you use more than one field/column for your SELECT, the DISTINCT will consider them all.

```
/*
Removes duplicates based on listed field(s)
*/

Let (
  [ $query = " SELECT DISTINCT s.amount
    FROM sales_related_sales AS s
    WHERE s.amount >= ?
    ORDER BY  s.amount DESC "

  ; $result = ExecuteSQL ( $query
    ; "" ; ""
    ; 490 )

  ]; $result
)
```

```
/*
Removes duplicates based on listed field(s)
*/

Let (
  [ $query = " SELECT DISTINCT lastname
    FROM myContacts
    WHERE LOWER(lastname) LIKE ?
    ORDER BY  lastname "

  ; $result = ExecuteSQL ( $query
    ; "" ; ""
    ; "ab%" )

  ]; $result
) // distinct lastname returns 192
// when distinct lastname, firstname is used,
// 2100 records are returned
```

HAVING is a Special WHERE for Aggregate fields

When you want to narrow your search based on the sum or count or other aggregate, the HAVING clause is used. Some SQL systems allow you to enter the aggregate in the SELECT statement and assign an alias to the new column. This can be used in the HAVING clause with just the alias. However, ExecuteSQL does not seem to allow this, so just repeat the aggregate in the HAVING clause, too, as shown in the examples below.

```
/*
HAVING used to filter
*/

Let (
  [ $query = " SELECT s.salespersonID
    , AVG( s.amount ) AS avg_amt
    FROM sales_related_sales AS s
    GROUP BY s.salespersonID
    HAVING AVG( s.amount ) > ? "

  ; $result = ExecuteSQL ( $query
    ; ","  ""
    ; 300 )

  ]; $result
) // SUCCESS - using "?" in select clause for a constant
```

```
/*
Using HAVING
*/

Let (
  [ $query = " SELECT '#'||s.salespersonID
    , SUM( s.amount ) AS sum_amount
    FROM sales_related_sales AS s
    GROUP BY s.salespersonID
    HAVING SUM( s.amount ) > 150000 "

  ; $result = ExecuteSQL( $query
    ; ","  "" )
```

```
   ]; $result
) // the HAVING clauses repeats the aggregate
```

JOINS & UNIONS

JOINS are what you do when you place two table occurrences on the Relationship Graph in FileMaker and connect two fields together. In SQL you simply name the two tables and add the relationship to the WHERE clause. The use of table ALIAS is more apparent when you start using joins. If you have more than one field/column with the same name, you must specify which table for each field, or you will get a SQL syntax error.

Implicit Joins

```
   FROM salesperson AS s, sales_related_sales AS sales
   WHERE s.salespersonID = sales.salespersonID
or
   FROM salesperson, sales_relates_sales
   WHERE filter_date = year_month
   // no need to use alias in the WHERE clause, as the fields/columns are unique
```

These are IMPLICIT JOINS. All tables are listed, the "join" is implied, and the simple equality is defined in the WHERE clause. Any of the comparison operators could be used, along with AND & OR. If you do not specify the WHERE clause at all, you get a Cartesian relationship. You probably rarely want Cartesian joins, but when you do (for returning global fields, perhaps?) an example is shown here:

Cartesian Join

```
/*
JOINS - cartesian - globals
*/

Let (
```

```
   [ $query = " SELECT
      s.name
      , s.salespersonID
      , global_num_g
      , global_txt_g
      FROM salesperson AS s, dev "

   ; $result = ExecuteSQL ( $query
      ; "," ; ""
      )

   ]; $result
) // no relationship between tables, so all records from both are returned
```

Inner Joins

INNER JOINS are the same as the implied equi-join (WHERE using the "=" matches, above), their syntax is slightly different.

```
   FROM salesperson AS s JOIN sales_related_sales AS sales
   ON s.salespersonID = sales.salespersonID
or
   FROM salesperson JOIN sales_related_sales
       ON filter_date = year_month
or
   FROM salesperson AS s
      INNER JOIN sales_related_sales AS sales
      ON s.salespersonID = sales.salespersonID
or
   FROM salesperson INNER JOIN sales_related_sales
       ON filter_date = year_month
```

They all work as well. The difference may be apparent if you have more than two files and must join them in a way that connects them accurately. An example would be Clients, Invoices & Invoice_items. Just as you put links between these tables on the FileMaker relationship graph, you link them with JOINs in SQL.

```
   FROM Clients AS c
      JOIN Invoices AS inv ON c.clientID_pk = inv.clientID_fk
```

```
        AND ( c.state = 'WA' OR c.state = 'ID' )
    JOIN Invoice_items AS itm
        ON inv.invoiceID_pk = itm.invoiceID_fk
        AND inv.invoice_date = '2009-10-13'
or
  FROM Clients AS c
    JOIN Invoices AS inv ON c.clientID_pk = inv.clientID_fk
    JOIN Invoice_items AS itm
        ON inv.invoiceID_pk = itm.invoiceID_fk
  WHERE ( c.state = 'WA' OR c.state = 'ID' )
    AND inv.invoice_date = '2009-10-13'
or (implied):
  FROM Client c, Invoices inv, Invoice_items inv
  WHERE c.clientID_pk = inv.clientID_fk
    AND inv.invoiceID_pk = itm.invoiceID_fk
    AND ( c.state = 'WA' OR c.state = 'ID' )
    AND inv.invoice_date = '2009-10-13'
```

The advantage may be the clear separation, so you understand what's being queried. But these should return the same results.

Outer Joins

OUTER JOINS are clearly different. Sometimes you want all of a "left-side" of a relationship AND any related records. This is similar to showing a portal on a layout. Find all records and some portals may be empty. This is a LEFT OUTER JOIN. You must specify the LEFT or RIGHT keyword when using OUTER JOIN. There are some articles on OUTER JOINS (see below).

```
  FROM Clients AS c
    LEFT OUTER JOIN Invoices AS inv ON c.clientID_pk = inv.clientID_fk
        AND ( c.state = 'WA' OR c.state = 'ID' )
    LEFT JOIN Invoice_items AS itm
        ON inv.invoiceID_pk = itm.invoiceID_fk
        AND inv.invoice_date = '2009-10-13'
// May return different records than just with "JOIN"
// or "INNER JOIN"!
```

Union

UNION is a keyword for combining two (or more) tables with the same number of fields in each SELECT statement. Perhaps you have archived records in a separate table and need to summarizes records from a current

and archived table in one report.

```
SELECT A, B, C, D
   FROM current_classes
   WHERE classID = 123
UNION ALL
SELECT a AS A, b AS B, c AS C, D
   FROM archive_classes
   WHERE classid = 456
ORDER BY D, B
// same number of columns/fields
// alias' used to name columns the same (not required)
// different find criteria (WHERE) is ok
// sort (ORDER BY) as the last clause
```

UNION (removes duplicates); UNION ALL (retains all records).

Tips, References & Resources

FileMaker Inc.

- ExecuteSQL, FM12 help topic
- FileMaker 12 ODBC and JDBC Guide (pdf)

General SQL Tutorials

- SQL.org Tutorial
- MySQL 5.6 Manual - 13.2.9. SELECT Syntax Not all the usage in MySQL translates for ExecuteSQL, but a handy reference, perhaps?
- w3schools - SQL Tutorial
- SQL Tutorial - SQL Query Reference
- sqlzoo - Interactive SQL tutorial *Examples*
- Built-in Functions (Transact-SQL) - MS SQL
- Chapter 4 SQL Functions - SQL in a Nutshell, O'Reilly
- "Sams Teach Yourself SQL in 10 Minutes" -Forta.

ExecuteSQL Blogs & Articles

- FM 12 ExecuteSQL, part 1 - Kevin Frank, 16 APR 2012
- FM 12 ExecuteSQL, part 2 - Kevin Frank, 22 APR 2012
- Outer Joins in FileMaker 12 - Part 1 - Kevin Frank, 2 OCT 2012

- Outer Joins in FileMaker 12 - Part 2 - Kevin Frank, 2 OCT 2012
- FM 12 ExecuteSQL: Dynamic Parameters, part 1 - Kevin Frank, 2 MAY 2012
- FM 12 ExecuteSQL: Dynamic Parameters, part 2 - Kevin Frank, 8 MAY 2012
- FM 12 ExecuteSQL: Robust Coding, part 1 - Kevin Frank, 13 MAY 2012
- FM 12 ExecuteSQL: Robust Coding, part 2 - Kevin Frank, 21 MAY 2012
- FM 12 ExecuteSQL "Unconference" Session - Kevin Frank, John Ahn, 19 JUL 2012 (11 SEP 2012: The ConditionalVL_SQL demo has been updated.)
- ExecuteSQL Function Notes - Kentuckiana FMpug
- Using ExecuteSQL to Query the Virtual Schema/System Tables - Andrew Duncan
- FileMaker 12: Why SQL? (Context Independent) - Seedcode, 6 APR 2012
- ExecuteSQL - Date Formats - Seedcode, 21 APR 2012
- Seedcode - search 'ExecuteSQL' - Seedcode blog
- FileMaker 12 SQL: Crafting Structural Beauty - Brian formats his ExecuteSQL so that it's a thing of beauty!

Helpful Databases and Custom Functions

- Skeleton Key webinar link to the YouTube video of the preso and download the example file used. YouTube
- SeedCode - SQLexplorer, Free tool for learning SQL in FM12. Include documentation and demonstration videos.
- Tip file: duplicating master and related records using ExecuteSQL and FM12 - G. Pupita The ExecuteSQL function introduced with FileMaker 12 now allows to create a completely modular procedure, a sort of "object" that can be re-used in any situation, just passing to the procedure tables and fields names. (examples in Italian or English for download)
- SQL Builder Eden Morris - More ambitious SQL generator, geared more to show all that can be done with SQL, so a steeper learning curve.
- FileMaker Pro 12 Adds Native SQL Queries - Anders Monsen, Mighty Data
- SQL.debug Custom function to return error instead of "?" - Andries Heylen
- Handy Custom Functions by Dan Smith - FMforums

FileMaker SQL Plug-ins

If you want to expand your knowledge of SQL and the additional commands you can make with plug-ins, try these:

- Dracoventions SQL Runner - SQL Runner is a free FileMaker Pro and Server database plug-in that lets you read and write data from and to FileMaker in powerful new ways.
- Youseful SQL Plugin - lets you run SQL queries directly on your FileMaker database.
- 360Works JDBC plugin - allows execution of arbitrary SQL statements on one or more JDBC databases, iterating result sets, and importing from any database which supports the JDBC protocol.
- 360Works Script Master - Direct access to the FileMaker SQL engine with ScriptMaster 4 Advanced, which allows you to execute SQL commands directly from any script.
- myFMbutler DoSQL plug-in - a FileMaker Pro plug-in for Windows and Macintosh that allows you to to manipulate FileMaker data from FileMaker calculations.
- MBS SQL Connections - contains functions to access SQL database servers directly.
- Goya - BaseElements Plugin - BE_FileMakerSQL
- CNS - MMQuery - MMQuery_ExecuteSQL: This function allows you to use SQL statements to run queries against the current Database File. MMQuery_ExecuteSQLEx: This function is only available in FileMaker Pro 11 and above. This "extended" version of MMQuery_ExecuteSQL allows you to optionally specify a separate, open Database File.
- SmartPill PHP edition - using the internal SQL functions (fm_sql_execute or fm_sql_select).

TIPS

- Format your "result" field with tab stops, if you are using Char(9) as your column/field delimiter in ExecuteSQL. If you know in advance the number of columns and approximate widths, you can set the tab stops with Inspector in Layout mode. You can also select all in the field in Browse mode and adjust the tab stops. You must View the Ruler to see these.
- The delimiters can be any legal characters. Some of the example files

use HTML tags as the delimiters. This is handy when the ExecuteSQL() is to be evaluated in a FileMaker Web Viewer. The default characters for the delimiters are comma and return. The demo file has an example of using concatenation & return as the field/column delimiter. This produces a "label-style" output.

- There is no TOP, LIMIT or ROWNUM for ExecuteSQL. You get all the results of the query. The suggestion has been made to use GetValue($result; 1), to show the first value returned if you use the default carriage return, Char(13), as the Row Delimiter.

Troubleshooting

If an error occurs during query parsing or execution, FileMaker Pro returns "?". Although not an argument (unless you disagree that you have entered your queries correctly!) it shows an error. The Error Codes are usually found in the FileMaker Help topic, FileMaker Pro 12 Error Code Reference Guide.

1. Try the query in one of the example databases first.
2. Paste the query into the Data Viewer (FileMaker Pro Advanced 12) to see any errors that might be generated.
3. Wrap the query in "EvaluationError (Evaluate ($query))" in the Data Viewer or as a Set Field to see any error codes returned.
4. There may be no errors, but the results don't appear as you expect. A SQL function may not be used by the ExecuteSQL at this time, for example.
5. It may be a formatting "error" that cannot be solved directly, or you need to change your delimiters.
6. Test the query in a Let() statement and assign variables as needed. If you use this in a Set Field script step, you can see what the variables are in the Data Viewer.
7. Two special error codes may be generated. These have been found by the Plug-in developers. "8309" = Semantics error (logical) & "8310" = Syntax Error (missing coding?).

```
/* sample query used in the enclosed demo */

Let (
```

```
  [ $query = "
    // your query here
    "

  ; $result = ExecuteSQL ( $query
    ; "" ; ""
    ; "" // optional arguments
    )

  ]; $result
) // notes here - you may add other variables for use with the final $result
```

DEMO FILES: SQL4_fmdev2.fmp12, related_sales.fmp12
A larger file with one million records was used for the contacts demos. It was too large to upload here, so you may import this smaller set: 4145Names