CHAPTER 26

Introducing ExecuteSQL

The *Structured Query Language (SQL)* is a standardized programming language that is used to manage relational databases and perform numerous operational functions to the data they store. SQL was initially created in the 1970s and soon became the standard programming language for relational databases. The American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) adopted an official SQL standard in 1986 and 1987 respectively. Since then, many updates to the standard have been released jointly by both organizations, with the most recent version adopted in December of 2011. Numerous companies now develop proprietary and open source SQL-compliant database systems.

In version 9.0 (2007), FileMaker introduced its first support for a SQL feature with the ability to create live connections to external ODBC data sources (Chapter 7). FileMaker version 12.0 (2012) introduced the `ExecuteSQL` function, adding the ability to perform SQL queries against FileMaker tables from any calculation formula within a FileMaker database.

A *SQL Query* is a text-based statement used to instruct a database to perform an action. The most frequently used type of query and the only one supported by the `ExecuteSQL` function in FileMaker is the `SELECT` query, which contains data retrieval instructions for a desired result set.

Experienced SQL programmers will appreciate the back-end functionality that this command adds to FileMaker but may find the function's limitation to the `SELECT` command constraining. Newcomers to database programming will find the function a little intimidating and divergent from FileMaker's more intuitive methods of data access. However, strictly from a FileMaker perspective, `ExecuteSQL` is like a miracle opening a new realm of possibility. The benefits of this function are immeasurable.

In FileMaker version 16, the `ExecuteSQL` function is matured. It allows a search, sort, and summarization of data to be performed directly from a calculation formula in a completely *context-independent* manner. It can perform a search in any table occurrence, including those that have no layout representation, and can even form temporary relationships dynamically. The functions it makes possible would otherwise require the creation of a multi-line script and extra table occurrences, relationships, and layouts to establish the necessary context to achieve the same result. Using this command, it is possible to severely limit the number of relationships required for data retrieval.

You don't need to learn SQL or use this command to create databases with FileMaker since the function is intended as a complement rather than a replacement to the more intuitive features available. However, once you realize what this function can do and how many resources it makes obsolete, you will want to use it everywhere it proves practical.

In this chapter, we will define and discuss the process of using the `ExecuteSQL` function, with topics such as:

- Defining the `ExecuteSQL` function
- Creating SQL queries
- Accessing the database schema

© Mark Conway Munro 2017
M. C. Munro, *Learn FileMaker Pro 16*, DOI 10.1007/978-1-4842-2863-0_26

■ **Tip** To experiment with the `ExecuteSQL` function, be sure to enter some test data into the database. To prepare the `Learn FileMaker` database for use with the examples in this chapter, import the `us-500.csv` download (Chapter 5) into the `Contact` table, matching up the name and address fields that we have already created.

Defining the ExecuteSQL Function

The `ExecuteSQL` function allows *any* calculation formula to retrieve data from *any* table occurrence within the relationship graph *completely independent* of any defined relationship or user interface context. Data can be accessed directly from the data table of any table occurrence without requiring hard-coded relationships, leaving the relationship graph less cluttered and convoluted.

A call to the `ExecuteSQL` function must include three parameters but can accept one or more optional `arguments` parameters, as shown in these two examples:

```
ExecuteSQL ( sqlQuery ; fieldSeparator; rowSeparator )
ExecuteSQL ( sqlQuery ; fieldSeparator; rowSeparator ; arguments )
```

The parameters in the statement are defined as follows:

- `sqlQuery` — a text expression or reference to a field that contains a `SELECT` statement.
- `fieldSeparator` — a text string containing the character(s) that should be used as a separator between fields in the result. If left empty, a comma will be used as a default separator.
- `rowSeparator` — a text string containing the character(s) that should be used as a separator between records in the result. If left empty, a carriage return will be used as a default separator.
- `arguments` — one or more text values that are used as dynamic parameters in the query, replacing question marks typically in a `WHERE` clause.

How the ExecuteSQL Function Works

The `ExecuteSQL` function works by accepting a `Select` statement that specifies which field(s) and record(s) should be extracted from a table and returned as text. It can optionally include delimiters to be inserted between fields (columns) and records (rows) in the result, as illustrated in Figure 26-1.

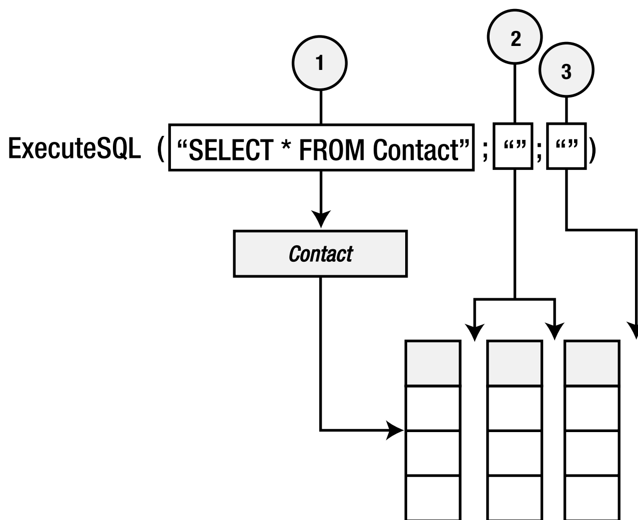

Figure 26-1. *An illustration of the basic functioning of the `ExecuteSQL` function, which specifies the data to be retrieved and optional delimiters to be inserted into that data*

The following three parameters are required by the function:

1. The `sqlQuery` parameter contains a `SELECT` query statement that informs FileMaker which field(s) to extract, which record(s) to include, and the table from which to extract them. The extracted information is in a temporary abstract columnar format that will be converted into a text result.
2. The optional `fieldSeparator` parameter allows you to define a delimiter, other than the default comma, which will be used between each field value returned when more than one field is requested.
3. The optional `rowSeparator` parameter allows you to define a delimiter other than the default carriage return, which will be used between each record.

The result is returned to the calling formula as a text string. When default delimiters are used, the string will take the form of a list of comma-separated field values in carriage-return separated records.

This example shows the formula pattern for requesting every field from a fictional table using default delimiters:

```
ExecuteSQL ( "Select * FROM Table" "", "" )
-- Result =
        Record1Field1,Record1Field2,Record1Field3,Record1Field4¶
        Record2Field1,Record2Field2,Record2Field3,Record2Field4¶
        Record3Field1,Record3Field2,Record3Field3,Record3Field4¶
```

This example shows a formula pattern for requesting the same information using an alternative `fieldSeparator`:

```
ExecuteSQL ( "Select * FROM Table" "--", "" )
-- Result =
        Record1Field1--Record1Field2--Record1Field3--Record1Field4¶
        Record2Field1--Record2Field2--Record2Field3--Record2Field4¶
        Record3Field1--Record3Field2--Record3Field3--Record3Field4¶
```

This example shows the formula pattern of a request using both an alternative `fieldSeparator` and `rowSeparator`:

```
ExecuteSQL ( "Select * FROM Table" "--", "#" )
-- Result =
        Record1Field1--Record1Field2--Record1Field3--Record1Field4#Record2Field1--
        Record2Field2--Record2Field3--Record2Field4#Record3Field1--Record3Field2--
        Record3Field3--Record3Field4#
```

Understanding Some Limitations of ExecuteSQL

Before moving on to actual examples, there are some limitations to the `ExecuteSQL` functions that should be understood to avoid frustration. These include:

- As mentioned in the introduction of this chapter, the `ExecuteSQL` function is currently limited to the `SELECT` command only. It is *not compatible* with other standard SQL functions that perform record changes or database schema modification such as `DELETE`, `INSERT`, `UPDATE`, `INSERT INTO`, `CREATE TABLE`, `DELETE TABLE`, etc.
- Relationships created in FileMaker are not recognized or required. However, a `SELECT` statement *can* use a `JOIN` clause to dynamically create temporary relationships for use within the query.
- The function does *not* recognize or require layout context. Instead it allows access to table records *directly* based on a named table occurrence.
- Values *must* be sent in an `SQL-92` compliant date and time formats with no braces to be properly recognized. To apply the correct formatting to dates and times in a query, use a `DATE`, `TIME`, or `TIMESTAMP` conditioning operator, preceding the string to have it properly recognized. Failure will cause the value to be evaluated as a literal string rather than a date or time. `ExecuteSQL` will *not* accept the `ODBC/JDBC` formats for `date`, `time`, and `timestamp` constants contained in braces.
- `ExecuteSQL` will return `date`, `time`, and `number` data using the `Unicode/SQL` format rather than the date and time settings of the database file or operating system.
- Sorting for SQL in FileMaker uses the `Unicode` binary sort order.
- `ExecuteSQL` is a *calculation function* and is not the same as the similarly named `Execute SQL` *script step*.

Creating SQL Queries

At minimum, the `sqlQuery` portion of the `ExecuteSQL` function requires a `SELECT` statement, which has numerous optional clauses available.

Defining SELECT Statements

The `ExecuteSQL` function's `sqlQuery` parameter must be a properly formatted `SELECT` statement. Minimally, it must contain an indication of *what* to find and from *where*, following this pattern:

```
SELECT <what> FROM <where>
```

The `<what>` is the name of one or more fields that should be selected and the `<where>` is the name of the table occurrence from which to extract them. Therefore, the pattern above can be formally expressed as the following formula:

```
SELECT <field(s)> FROM <table>
```

For example, this statement would return the contents of a field named `Name` from every record in a table occurrence named `Contact`:

```
SELECT Name FROM Contact
```

The above formula defines a `SELECT` statement requesting *all records* without further processing. In addition to this basic requirement, the statement can include many optional clauses that expand the functionality of the query and offer refinement of and manipulation to the results. The `SELECT` command makes available features such as:

- `SELECT` — Request one or more fields from one or more tables.
- `SELECT DISTINCT` — Eliminate duplicates from the result.
- `JOIN` — Create temporary relationships.
- `WHERE` — Find sets of records matching specific criteria.
- `AS` — Create dynamic aliases of tables and fields to shorten the length of clauses.

The full formula, including every optional clause except aliasing, is:

```
SELECT/SELECT DISTINCT <fields>
FROM <tables>
JOIN <table> ON <formula>
WHERE <formula>
GROUP BY <fields>
HAVING <formula>
UNION <select>
ORDER BY <fields>
OFFSET <number> ROW/ROWS
FETCH FIRST <number> PERCENT/ROWS/ROW/ONLY/WITH TIES
```

Each available clause is defined in Table 26-1.

Table 26-1. The definitions of each available clause of a SELECT statement

Keyword	Clause Description
SELECT	Specifies one or more fields to select. Can include fields, constants, calculations, and functions. Use an asterisk to select all fields.
SELECT DISTINCT	Adding the DISTINCT operator will remove any duplicates from the result.
FROM	Specifies one or more tables from which to select the fields.
JOIN	Contains a table and relationship formula to allow for the extraction of fields through a temporarily established relationship.
WHERE	Contains one or more formulas that specify the criteria that all results must match, like filtering the results based on search criteria.
GROUP BY	Specifies one or more fields to use to summarize the results.
HAVING	Contains one or more formulas that specify the criteria for the inclusion of a summarized result. HAVING is to a GROUP BY what a WHERE is to a SELECT.
UNION	Used to combine two or more SELECT statements into a single result.
ORDER BY	Specifies one or more fields to use to sort the results.
OFFSET	Used to specify the number at which results should begin, thereby excluding a set of records prior to that number.
FETCH FIRST	A number specifying the how many rows that should be retrieved from the starting point.
AS	Used to create a shorter alias for a table name that can be used in its place elsewhere in the statement as a prefix to identify a field's table, especially when there is more than one table involved as when using a JOIN clause.

Each of these operational clauses are woven into a text string with other combinations of literals, keywords, table occurrences, and field names to form a SELECT SQL query, which can be passed as the first parameter to the ExecuteSQL function.

Formatting Requirements

There are a few formatting requirements to keep in mind when writing SELECT statements for the ExecuteSQL function. These include:

- *Command and Object Names* — The names of tables, fields, and even statement commands within a query are *not* case sensitive. For example, using either select or SELECT will work the same. However, using capitalization for all commands and operators can help to differentiate them from the value of specific tables, fields, and other variable information in the query. The examples in this chapter will adhere to that approach.
- *Criteria* — Literal criteria, such as that used within a JOIN, WHERE, and HAVING clause, *is* case sensitive and will fail to locate matching values of a different case. Also, all textual criteria *must* be enclosed in single quotations.

- *Name Separators* — When listing multiple tables and fields, always use a comma-space delimiter between them.
- *Quotations* — Table and field names don't need to be enclosed in double quotations *unless they contain spaces*. Table names that begin with non-alphabetic characters must be enclosed in double quotations even when they don't contain spaces.

Using the SELECT Statement

Although the `SELECT` statement is the only one supported by the `ExecuteSQL` function in FileMaker, it is quite powerful and flexible. Before delving into the many different optional clauses, let's explore the basic statement's capabilities and discuss techniques for using it effectively.

Selecting an Entire Table

The most basic SQL query is one in which every field/column will be selected for every record/row. This can be performed with a simple statement:

```
SELECT * FROM <Table>
```

The statement must *always* begin with the word `SELECT` since that is the only primary function currently supported by FileMaker. After this the statement must indicate which fields to select and `FROM` which table. In this case, the asterisk informs FileMaker to select *all fields*. The `<Table>` placeholder is replaced with the name of an actual table occurrence whose base table the function should access. Change this portion of the formula to `Contact` and the following `SELECT` statement we fetch every field from the `Contact` table will read as follows:

```
SELECT * FROM Contact
```

This `SELECT` statement becomes the first parameter of the `ExecuteSQL` statement:

```
ExecuteSQL ( "SELECT * FROM Contact" ; "" ; "" )
```

Open the `Learn FileMaker` database and enter this as the formula for the `Example Calculation` field. After confirming that the `Calculation result` type is of `Text`, save the formula, and the result displayed in Figure 26-2 should appear in the field. Each paragraph of the result is a single record that contains a comma-separated list of field values. Both records and fields are displayed in creation order.

```
000001,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,James,Butt,,,6649 N Blue Gum St,New Orleans,70116,Orleans,LA
000002,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Josephine,Darakjy,,,4 B Blue Ridge Blvd,Brighton,48116,Livingston,MI
000003,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Art,Venere,,,8 W Cerritos Ave #54,Bridgeport,08014,Gloucester,NJ
000004,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Lenna,Paprocki,,,639 Main St,Anchorage,99501,Anchorage,AK
000005,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Donette,Foller,,,34 Center St,Hamilton,45011,Butler,OH
000006,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Simona,Morasca,,,3 Mcauley Dr,Ashland,44805,Ashland,OH
000007,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Mitsue,Tollner,,,7 Eads St,Chicago,60632,Cook,IL
000008,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Leota,Dilliard,,,7 W Jackson Blvd,San Jose,95111,Santa Clara,CA
000009,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Sage,Wieser,,,5 Boston Ave #88,Sioux Falls,57105,Minnehaha,SD
000010,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Kris,Marrier,,,228 Runamuck Pl #2808,Baltimore,21224,Baltimore City,MD
000011,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Minna,Amigon,,,2371 Jerrold Ave,Kulpsville,19443,Montgomery,PA
000012,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Abel,Maclead,,,37275 St  Rt 17m M,Middle Island,11953,Suffolk,NY
000013,15:30:14,2016-11-04,2016-11-04,3:30:14 PM,Mark Munro,Mark Munro,,Kiley,Caldarera,,,25 E 75th St #69,Los Angeles,90034,Los Angeles,CA
```

Figure 26-2. *An example result of the `ExecuteSQL` function when requesting every field from every record of the `Contact` table*

■ **Caution** Any table names that include spaces or begins with non-alpha characters *must* be enclosed in double quotations to work correctly in an SQL query. Since the `Select` statement is contained in quotes within the `ExecuteSQL`, the inner quotations around the table name must be escaped (`backslash-quote`) as shown in this example showing a hypothetical table named `Expense Data`:

```
ExecuteSQL ( "SELECT * FROM \"Expense Data\"" ; "" ; "" )
```

Selecting Individual Fields

While there may be situations where a formula requires access to *every* field for *every* record in a table, most of the time you will be selecting data with more specificity. Fortunately, the `SELECT` statement allows the selection of one or more individual fields, producing a more focused result.

■ **Caution** Like table names, field names that contain spaces or reserved words must be enclosed in double quotations.

Specifying a Single Field

To select a *single field* from a `Contact` table, change the asterisk to the name of the field. The formula for this would be:

```
SELECT <Field> FROM <Table>
```

To select the `Contact Address City` field from the `Contact` table, write the following formula:

```
ExecuteSQL ( "SELECT \"Contact Address City\" FROM Contact" ; "" ; "" )
```

This statement will result in a carriage return-delimited list of city names from every record in the table, partially shown in Figure 26-3. Notice that the list includes the field value for *every* record, so it will include many duplicates like those highlighted.

New Orleans
Brighton
Bridgeport
Anchorage
Hamilton
Ashland
Chicago
San Jose
Sioux Falls
Baltimore
Kulpsville
Middle Island
Los Angeles
Chagrin Falls
Laredo
Phoenix
Mc Minnville
Milwaukee
Taylor
Rockford
Aston
San Jose
Irving
Albany
Middlesex
Stevens Point
Shawnee
Easton
New York

Figure 26-3. *The list of city names selected from the* `Contact` *table*

Specifying Multiple Fields

To select multiple fields from the `Contact` table with a single query, list each field in a comma-space separated string. The formula for this would be:

```
SELECT <Field1>, <Field2>, <Field3> FROM <Table>
```

To select the `Contact Address City` and `Contact Address State` fields from the `Contact` table, write the following formula:

```
ExecuteSQL (
        "SELECT \"Contact Address City\", \"Contact Address State\" FROM Contact" ;
        "" ; "" )
```

The result of this statement will be a comma-delimited list of city and state names, partially shown in Figure 26-4.

New Orleans,LA
Brighton,MI
Bridgeport,NJ
Anchorage,AK
Hamilton,OH
Ashland,OH
Chicago,IL
San Jose,CA
Sioux Falls,SD
Baltimore,MD
Kulpsville,PA
Middle Island,NY
Los Angeles,CA
Chagrin Falls,OH
Laredo,TX
Phoenix,AZ
Mc Minnville,TN
Milwaukee,WI
Taylor,MI
Rockford,IL
Aston,PA
San Jose,CA
Irving,TX
Albany,NY
Middlesex,NJ
Stevens Point,WI
Shawnee,KS
Easton,MD
New York,NY

Figure 26-4. *The list of cities and states selected from the* `Contact` *table*

Keeping References Dynamic

When building a query, it is a good idea to avoid typing the name of tables and fields as static text and instead use dynamic references. Then, if the name of a table or field changes in the future, the query will continue to function without modification since the reference to the field updates automatically.

The following formula demonstrates a simple example of this technique:

```
Let ( [
        reference = GetFieldName ( Contact::Contact Name First ) ;
        reference = Substitute ( reference ; "::" ; "¶" ) ;
        table.name = GetValue ( reference ; 1 ) ;
        field.name = Quote ( GetValue ( reference ; 2 ) )
] ;
        ExecuteSQL ("SELECT " & field.name & " FROM " & table.name ; "" ; "" )
)
```

The `Let` statement converts the dynamic field reference into a text string and places it into the `reference` variable. The `Substitute` function is then used to replace the two colons with a paragraph return, placing the following text value into the `reference` variable:

```
Contact
Contact Name First
```

Finally, each value is extracted and placed into the `table.name` and `field.name` variables respectively. From there, they can be used to construct the `SELECT` statement that is placed into an `ExecuteSQL` function.

This extra effort ensures that any changes to the table or field name will automatically be updated and the query will continue to function as expected.

Using SELECT DISTINCT to Return Unique Values

To automatically eliminate duplicate values and alphabetize the results, use the `SELECT DISTINCT` command, as shown in this formula:

```
SELECT DISTINCT <Field> FROM <Table>
```

The following formula will generate a list of alphabetically sorted, unique values from the `Contact Address City` field of the `Contact` table:

```
ExecuteSQL ( "SELECT DISTINCT \"Contact Address City\" FROM Contact" ; "" ; "" )
```

The uniqueness of the list is based on the *entire* record value of the result, not individual fields within it. For example, when selecting only the city, the results will include only one entry for San Jose. However, if multiple fields are selected, like the street address and city, as shown in the example below, the results will include multiple entries for San Jose since the rest of the record/row includes other values.

```
ExecuteSQL (
"SELECT DISTINCT \"Contact Address Street\", \"Contact Address City\" FROM Contact"
; "" ; "" )
```

Since `123 First Street,San Jose` is not equal to `555 Selma Ave,San Jose`, both will appear in the result.

Reformatting SELECT Statements for Clarity

The preceding examples are intentionally short and easy to read. However, a `SELECT` statement can and will grow and begin wrapping onto multiple lines due to the following:

- Additional `SELECT` clauses.
- Dynamic construction of a `SELECT` clause using FileMaker resources: field references, functions, and variables.
- Dynamic construction of a `SELECT` clause with variable components.
- Requesting multiple fields in a single clause.

There are two basic methods one can employ to reformat the statements to improve clarity and readability: adding carriage returns and tabs or using a `Let` statement.

Adding Carriage Returns and Tabs

FileMaker will ignore carriage returns and tabs in the sqlQuery text string so these can be used to separate the statement into hierarchically readable blocks. Doing so will break up the continuous text flow and make the statement easier to read and troubleshoot, as shown here:

```
ExecuteSQL ( "
        SELECT <field>
        FROM <table>
        JOIN <table> ON <formula>
        WHERE <formula>
        ORDER BY <field>
" ; "" ; "" )
```

In the formula above, a carriage return is used to separate the actual query from the enclosing ExecuteSQL statement. Also, a combination of a carriage return and tab is used to separate each clause of the statement onto its own line, where it is indented to stand out from the enclosing statement.

When multiple fields, find conditions, or other components are used, those can be pushed onto their own line, further indented for additional clarity as shown here:

```
ExecuteSQL ( "
        SELECT
                <field1>,
                <field2>,
                <field3>
        FROM <table>
        JOIN <table> ON <formula>
        WHERE
                <condition1> and
                <condition2> and
                <condition3>
        ORDER BY
                <field1>,
                <field2>
" ; "" ; "" )
```

Using a LET Statement

Another method of eliminating the visual confusion of a complex query is the use of a LET statement. The entire SELECT query can be built in pieces using separate variables and then combined into a single variable before being inserted into the ExecuteSQL statement, as shown here:

```
Let ( [
        s.Fields = "SELECT <Fields>" ;
        s.Table = "FROM <Table> " ;
        s.Join = "JOIN <table> ON <formula> " ;
        s.Where = "WHERE <formula> " ;
        s.Group = "GROUP BY <fields> " ;
```

```
        SQL = s.Fields & s.Table & s.Join & s.Where & s.Group
] ;
        ExecuteSQL ( SQL ; "" ; "" )
)
```

Exploring the Benefits of Aliases

In a SQL query, an *alias* is a short text string made up of one or more characters that can act as a proxy for a table name elsewhere in a `SELECT` statement. When a `SELECT` statement contains references to more than one table, as in a `JOIN` clause, aliases are used to identify the table to which a field belongs.

Although an alias can be made up of one or more characters, as a space-saving mechanism, shorter is better. To establish an alias, use the `AS` clause after the identification of a table and follow it with a text alias, as shown here:

```
SELECT <field> FROM <table> AS <alias>
```

For example, to create an alias named `c` for the `Contact` table, the `FROM` clause would appear like this:

```
FROM Contact AS c
```

Once the alias `c` is established, it can be used as a prefix on any field name, in any clause, to identify the table to which a field belongs. While aliases are not required when selecting fields from a single table, this example illustrates that it is possible.

```
SELECT c.Notes FROM Contact AS c
```

■ **Note** Although the `AS` clause establishes the alias at the end, it can be used *anywhere* within the statement.

Again, in such a simple statement, using an alias is unnecessary as illustrated with these three examples that all generate the exact same result:

```
SELECT Notes FROM Contact
SELECT Contact.Notes FROM Contact
SELECT c.Notes FROM Contact AS c
```

When a field with spaces in its name is enclosed in double quotations, the alias prefix should precede the name, outside of the quotes, as shown here:

```
SELECT c.\"Contact First Name\" FROM Contact AS c
```

Inserting Literal Text in the Field List

Literal text strings can be inserted before, between, or after field names within the `SELECT` statement and will be repeated in the results for each record. Literals must be enclosed in single quotation marks anywhere they are used.

■ **Note** Literals are also used in clauses that include formulas such as `WHERE` and `JOIN`, which are discussed later in this chapter.

This example demonstrates a literal value inserted into the field results as a field label:

```
ExecuteSQL ( "
	SELECT
		'Name: ',
		\"Contact Name First\",
		\"Contact Name Last\"
	FROM Contact"
; " " ; "" )
```

By placing a literal `'Name:'` with the field names and using a `fieldDelimiter` of a space in the second parameter of the `ExecuteSQL` function, this example will return a list of names with a preceding label:

```
Name: Josephine Darakjy
Name: Art Venere
Name: Shannon Miller
```

Concatenating Results

Concatenation is the action of linking things together in a chain or series and is possible within SQL queries. Instead of receiving a raw comma-delimited set of field values and then parsing and manipulating that information with other formulas or scripts, concatenation allows pre-processing of field values into more useful results *directly within a query statement*.

Concatenation of text values is achieved using either the + or || operators x. However, the latter has been reportedly more reliable and may be less likely to be confused with the same operator used in mathematical calculations. The code below shows an example of concatenation of a contact's first and last names into a single string, with a literal space inserted between them. This is achieved by replacing the comma-space between the fields with a `space`, `double-bar`, `space`, `single quote`, `space`, `single quote`, `space`, `double-bar`, `space`, as shown here:

```
ExecuteSQL ( "
	SELECT \"Contact Name First\" || ' ' || \"Contact Name Last\"
	FROM Contact
" ; "" ; "" )
```

Using the alternative plus-sign delimiter, the `SELECT` line of the above example, would be:

```
SELECT \"Contact Name First\" + ' ' + \"Contact Name Last\"
```

In both examples, each record of the result should contain the first and last name of a contact with a space between them. For example, with concatenation, the examples would return a result of `Art Venere` instead of `Art,Venere`, without it. While the same result could have been achieved using the `fieldDelimiter` parameter, by concatenating the fields directly in the statement, that delimiter remains available for use between other fields that may be included in the query but not concatenated within the same statement.

This example shows how to return a contact's full name and entire mailing address as a continuous concatenated string:

```
ExecuteSQL ( "
        SELECT
                \"Contact Name First\" + ' ' +
                \"Contact Name Last\" + ' ' +
                \"Contact Address Street\" + ' ' +
                \"Contact Address City\" + ', ' +
                \"Contact Address State\" + ' ' +
                \"Contact Address Zip\"
        FROM Contact
" ; "" ; "" )
```

Using the WHERE Clause

The `WHERE` command can be added to a `SELECT` statement to specify criteria for the records that should be included in the result, as shown here:

```
SELECT <field> FROM <table> WHERE <formula>
```

The `<formula>` portion of the clause must contain one or more expressions that specify the criteria used to qualify records that should be returned. These will typically include a field, operator, and either a literal value or another field.

Creating a Single Expression WHERE Clause

To select `Contact` records with a California address, the expression of the formula would be:

```
"Contact Address State" = 'CA'
```

■ **Note** Just like when using fields in the select list, fields in the `WHERE` clause must be enclosed in double quotations if they contain spaces. Also, literal textual values must always be enclosed in single quotations.

The following shows an example of a `SELECT` statement with this expression in a `WHERE` clause to extract the first and last name of every contact with an address in California from a `Contact` table:

```
SELECT
        "Contact Name First",
        "Contact Name Last"
FROM Contact
WHERE "Contact Address State" = 'CA'
```

Creating a Multiple Expression WHERE Clause

When using multiple expressions, they must be separated by a comparison operator of AND or OR.

For example, when searching for contacts living in a city that is common to many states, such as Milford, you might need two expressions to specify both the city *and* the state. In that case, you would use the AND operator between the two expressions, as shown in this example:

```
WHERE "Contact Address City" = 'Milford' AND "Contact Address State" = 'PA'
```

Similarly, to find contacts from two different states, for example, from Pennsylvania *or* Ohio, you would use an OR operator, as shown in this example:

```
 WHERE "Contact Address State" = 'PA' OR "Contact Address State" = 'OH'
```

Using the ORDER BY Clause

The ORDER BY clause can be added to a SELECT statement to specify the field(s) by which the results should be sorted, as shown in this formula:

```
SELECT <field> FROM <table> ORDER BY <fields>
```

For example, to return a list of every contact's last name sorted by state:

```
SELECT "Contact Name Last" FROM Contact ORDER BY "Contact Address State"
```

Combining the ORDER BY with a WHERE clause, this example will return the last name of every contact living in a city named "Milford" sorted by state:

```
SELECT "Contact Name Last"
FROM Contact WHERE "Contact Address City" = 'Milford'
ORDER BY "Contact Address State"
```

Using the JOIN Clause

A JOIN clause can be added to a SELECT statement to create a temporary relationship between two table occurrences. Joining tables is useful when selecting fields from both a local and related table, blending them into a single result. Joins allow other clauses with field references to use fields from either or both tables. For example, a WHERE or ORDER BY clause can each contain fields from either table, once joined.

The JOIN clause must contain the name of a table that should be related to the FROM table and include a formula containing one or more expressions that specify the criteria used to connect records into a temporary relationship, as shown in this formula:

```
SELECT <field> FROM <table1> JOIN <table2> ON <formula>
```

To connect the `Contact` and `Company` tables in the `Learn FileMaker` database, assign them an alias of `con` and `com` respectively and add a JOIN clause. To retrieve the `Contact Name First` field and the related `Company Name` field, use the following query:

```
SELECT
        con."Contact Name First",
        com."Company Name"
FROM Contact AS con
JOIN Company AS com ON
        con."Contact Company ID" = com."Record ID"
```

The example above declares aliases in the `SELECT` and `JOIN` statements and then uses them throughout the statement.

■ **Tip** The `JOIN` clause can also be used to create self-joins, which is a table related to itself.

Using the GROUP BY Clause

The `GROUP BY` clause can be added to a `SELECT` statement to generate an aggregate value based on one or more fields, as shown in this formula:

```
SELECT <fields> FROM <table1> GROUP BY <field>
```

The clause acts much like a FileMaker-native summary field when placed in a sub-summary part on a layout displaying records sorted by the appropriate field. It generates a summarization of data based on a sort/grouping field.

For example, if the `Contact` table had a field called `Invoices` that contained a total of a customer's invoices, the following code *without* a `GROUP BY` clause will return a list of each *contact's* `State` and their `Invoice` amount:

```
SELECT State, Invoices FROM Contact ORDER BY State
-- Result =
AK,1000
AK,500
AK,250
AZ,500
AZ,750
Etc.
```

By contrast, using the `Sum` function on the `Invoices` field and adding a `GROUP BY` clause to group by state, this example will return a summary of invoices by state:

```
SELECT State, Sum ( Invoices ) FROM Contact GROUP BY State
-- Result =
AK,1750
AZ,1250
Etc.
```

Notice that the ORDER BY clause is removed from the example above since the GROUP BY clause will automatically sort the results by the grouping field(s).

Adding a HAVING Clause

The HAVING clause can be added to a GROUP BY clause of a SELECT statement to filter the results based on a formula:

```
SELECT <fields> FROM <table> GROUP BY <field> HAVING <formula>
```

The clause acts like a WHERE clause but for grouped results. Using the previous example, we can add a HAVING clause to only include states where the summary of Invoices is greater than a certain dollar amount, as shown here:

```
SELECT State, Sum ( Invoices ) FROM Contact GROUP BY State HAVING Sum ( Invoices ) > 1500
-- Result =
AK,1750
CA,2500
Etc.
```

In the results, the summarized entry for AZ has been removed because the summary of 1250 was under the threshold of 1500 that is specified in the HAVING clause.

Using the UNION Clause

The UNION clause will combine the results of two or more SELECT statements, whether from the same table or from different tables, if two conditions are met:

- Each SELECT statement must select the same number of fields.
- Each field position must return the same data format across all SELECT statements.

For example, if the first SELECT statement returns three fields with the data types of text, number, and text, the second SELECT statement must also return three fields with the same data types in the same order.

The formula for the UNION clause is:

```
SELECT <fields1> FROM <table1> UNION SELECT <fields2> FROM <table2>
```

By default, the clause will automatically exclude duplicate entries from the merged result. Use UNION ALL to include all results, even duplicates, for example:

```
SELECT <fields1> FROM <table1> UNION ALL SELECT <fields2> FROM <table2>
```

Limiting the Results of a Query

The OFFSET and FETCH FIRST clauses can be used separately or in unison to control the number of results returned by a query.

Using the OFFSET Clause

The OFFSET clause can be added to a SELECT statement to indicate the number of records/rows in the result to skip over, as shown in this formula:

```
SELECT <fields> FROM <table> OFFSET n ROWS
```

This example will return a list of all the Record ID field values from the Contact table, starting from record 21:

```
SELECT \"Record ID\" FROM Contact OFFSET 20 ROWS
```

Using the FETCH FIRST Clause

The FETCH FIRST clause can be added to a SELECT statement to limit the number of rows returned, as shown in this formula:

```
SELECT <fields> FROM <table> FETCH FIRST n ROWS ONLY
```

This example will return the Record ID for the first 10 records:

```
SELECT \"Record ID\" FROM Contact FETCH FIRST 10 ROWS ONLY
```

Combining the OFFSET and FETCH FIRST Clauses

When combined, the OFFSET and FETCH FIRST clauses allow you to access groups of results one batch after the other, a process often referred to as "paging" through results, since each group is presented as the next "page" in a list of results.

```
SELECT <fields> FROM <table> OFFSET n ROWS FETCH FIRST n ROWS ONLY
```

The OFFSET portion indicates where the desired group starts and the FETCH FIRST portion limits the number of records accessed from that starting point. This code shows several examples of accessing batches of records ten at a time:

```
SELECT \"Record ID\" FROM Contact FETCH FIRST 10 ROWS ONLY
SELECT \"Record ID\" FROM Contact OFFSET 10 ROWS FETCH FIRST 10 ROWS ONLY
SELECT \"Record ID\" FROM Contact OFFSET 20 ROWS FETCH FIRST 10 ROWS ONLY
SELECT \"Record ID\" FROM Contact OFFSET 30 ROWS FETCH FIRST 10 ROWS ONLY
SELECT \"Record ID\" FROM Contact OFFSET 40 ROWS FETCH FIRST 10 ROWS ONLY
```

The first statement will return records 1 through 10. The second statement will return records 11 thru 20, the third will return records 21 thru 30 and so on.

Accessing the Database Schema

One of the lesser known uses of the ExecuteSQL function is the ability to access two "virtual tables," hidden in every FileMaker database that provide information about the scheme. These are:

- FileMaker_Tables
- FileMaker_Fields

These tables can be used in a SELECT statement just like a custom table.

Selecting FileMaker_Tables

The FileMaker_Tables table contains one record for every table occurrence defined in the relationship graph with the following fields of information:

- TableName — The name of the table occurrence.
- TableID — FileMaker's identification number for the table occurrence is unique within the database.
- BaseTableName — The name of the actual table upon which the table occurrence is based.
- BaseFileName — The name of the file in which the table occurrence's base table exists.
- ModCount — The number of modifications made to the table.

Each of these fields can be used in a SELECT statement just like a custom field from your tables. To get all five fields for every table in the database, use the asterisk, as shown here:

```
ExecuteSQL ( "SELECT * FROM FileMaker_Tables" ; "" ; "" )
        -- Result =
        Company,1065101,Company,Learn FileMaker,20
        Company | Contact,1065105,Contact,Learn FileMaker,24
        Contact,1065102,Contact,Learn FileMaker,24
        Contact | Company,1065106,Company,Learn FileMaker,20
        Project,1065103,Project,Learn FileMaker,4
        Project | Company,1065107,Company,Learn FileMaker,20
        Sandbox,1065089,Sandbox,Learn FileMaker,134
```

To get a list of every table occurrence in the database, specify the TableName field only:

```
ExecuteSQL ( "SELECT TableName FROM FileMaker_Tables" ; "" ; "" )
        -- Result =
        Company
        Company | Contact
        Contact
        Contact | Company
        Project
        Project | Company
        Sandbox
```

To limit the results to the actual table names, use SELECT DISTINCT and request the BaseTableNamee:

```
ExecuteSQL ( "SELECT DISTINCT BaseTableName FROM FileMaker_Tables" ; "" ; "" )
        -- Result =
        Company
        Contact
        Project
        Sandboxe
```

■ **Tip** All the clauses and options available in the SELECT statement can be used when accessing the hidden FileMaker_Tables and FileMaker_Fields tables.

Selecting FileMaker_Fields

The FileMaker_Fields table contains one record for every field defined in the database file with the following fields of information, all accessible through the ExecuteSQL function:

- TableName — The name of the table occurrence whose base table contains the field exists.
- FieldName — The name of the field.
- FieldType —The SQL data type of the file.
- FieldID — FileMaker's identification number for the field is unique within the base table containing the field.
- FieldClass — The class of the field: Normal, Summary, or Calculated.
- FieldReps — The number of maximum repetitions defined for the field.
- ModCount — The number of modifications made to the table.

Each of these fields can be used in a SELECT statement just like any custom field from your tables. To get all seven fields for every field in the database, use the asterisk, as shown here:

```
ExecuteSQL ( "SELECT * FROM FileMaker_Fields" ; "" ; "" )
```

Since the hidden fields table contains fields based on table occurrences, the results will include duplicates if there is more than one occurrence for a given base table. A custom script or a recursive custom function like the one below named Unique_FileMaker_Fields can be used to get a list of the name of every base table and then retrieve the fields for each, resulting in a clean list of base tables and their fields without any duplicates:

```
Let ( [
        current.table = GetValue ( baseTables ; 1 ) ;
        baseTables = RightValues ( baseTables ; ValueCount ( baseTables ) - 1 )
] ;
        ExecuteSQL (
                "SELECT * FROM FileMaker_Fields WHERE TableName='" & current.table & "'"
                ; "" ; ""  ) &
        Case ( baseTables ≠ "" ; Unique_FileMaker_Fields ( baseTables ) )
)
```

Exploring Other SQL Features

FileMaker's SQL functionality is a lot more feature rich than the material covered in this chapter. For example:

- Numerous functions can be embedded into a `SELECT` statement to manipulate the results, many providing functionality like FileMaker's own functions. These include functions for:
  - Date and time manipulation
  - String manipulation
  - Aggregating numeric values
  - Performing mathematical computations
  - Taking conditional action
- Numerous operators can be used with field values and SQL functions to manipulate results within the `SELECT` statement.
- The `Execute SQL` script step allows more robust manipulation of external `ODBC/JDBC` data sources from external databases.
- FileMaker can be used as an `ODBC/JDBC` data source and supports SQL queries from external databases.

For more information about these topics, visit `filemaker.com` and look for these and other guides: The *FileMaker ODBC and JDBC Guide* is available at:

`https://fmhelp.filemaker.com/docs/16/en/fm16_odbc_jdbc_guide.pdf.`

The *FileMaker SQL Reference* is available at:

`https://fmhelp.filemaker.com/docs/16/en/fm16_sql_reference.pdf.`

Summary

In this chapter, we introduced the `ExecuteSQL` function and discussed many features of the `SELECT` statement.