

The Basics of Querying in FoxPro with SQL SELECT

(Lesson I Single Table Queries)

The idea behind a query is to take a large database, possibly consisting of more than one table, and producing a “result set”, which is a smaller table of rows and columns reflecting a subset of the overall database and which can be used to provide one or more answers to user needs. In FoxPro, this need is frequently met using the powerful SQL SELECT command.

The complete SELECT syntax is:

```
SELECT [ALL | DISTINCT]
      [<alias>.]<select_item>
          [AS <column_name>]
          [, [<alias>.]<select_item>
              [AS <column_name>] ...]
FROM <table> [<local_alias>]
    [, <table> [<local_alias>] ...]
[[INTO <destination>]
    | [TO FILE <file>
        [ADDITIVE]
        | TO PRINTER [PROMPT]
        | TO SCREEN]]
[PREFERENCE <name>]
[NOCONSOLE]
[PLAIN]
[NOWAIT]
[WHERE <joincondition>
[AND <joincondition> ...]
[AND | OR <filtercondition>
[AND | OR <filtercondition> ...]]]
[GROUP BY <groupcolumn>
    [, <groupcolumn> ...]]
[HAVING <filtercondition>]
[UNION [ALL] <SELECT command>]
[ORDER BY <order_item>
    [ASC | DESC]
    [, <order_item>
    [ASC | DESC] ...]]
```

That’s a great way to start out a “basic” lesson, right? Actually, I included it on Page 1, because you’ll frequently want it handy as a reference.

To simplify the above as a starting point, the FoxPro command syntax places optional clauses in square brackets. Thus, the most basic form of the command is:

```
SELECT <select_item>
      FROM <table>
```

The **table** argument is always the name of a database table (or .DBF), and the most frequently-used **select_item** is a field name from that table. An example of such a statement is:

1) SELECT Name; FROM Asset	Selects the Name column (field) of all records from the Asset table.
-------------------------------	--

This query obviously is not very useful, since you could have just opened the Asset table and issued a "BROWSE FIELDS Name" command. Nevertheless, the SELECT approach consists of a single statement (instead of two), and works *irrespective of whether the table was previously open*. As you will soon see, this first SELECT statement only needs minor adornment to become a useful query.

A WORD ABOUT SYNTAX

The code examples shown here depict the **SELECT** statements on multiple lines. This is done for ease of reading, and is recommended for use in your actual code.

Note that for a single command to occupy multiple textual lines, Fox Pro requires a semi-colon ";" at the end of all but the final line. This is the "line-continuation" indicator.

TIP: You can type multiple line commands from the Command Window by using {Ctrl}+{Enter} to insert a hard line break without instead attempting to execute the command on the first line. Once done typing the entire command, simply press {Enter} with the cursor on any of the lines to execute the command.

The first clause in our SELECT statement is the **select_item**, and there are two important things to know. The first is that there can be more than one item selected. Each is separated from the previous by a comma (.). Make sure not to include a comma after the last item or FoxPro will try to interpret the first word of the next clause (e.g., "FROM") as a field name! Thus we could easily expand our query to:

2) <code>SELECT Name, Group ; FROM Asset</code>	Selects the Name <u>and</u> Group columns of all records from the Asset table.
---	--

The second important aspect of a **select_item** is that it does not always have to be a field Name. It can be any of the following:

- a valid field from the specified table (as in our first examples);
- an asterisk (*), which indicates to select all fields from the table;
- a FoxPro expression that can be properly evaluated according to FoxPro's SQL rules (too complex to explain further here); or
- any of several SQL statistical functions, the most common of which are COUNT() and SUM().

Each of these forms will be used in examples in the remainder of this document. Expressions, fields, and statistical functions can be mixed within a list of selected columns. Note that using any statistical SQL function completely changes the nature of what the SELECT statement does. (See next section.) The following examples show typical usages:

3) <code>SELECT* ;</code>	Selects all fields of all records from
---------------------------	--

FROM Asset	the Asset table.
4a) SELECT Name, LEFT(Manager, 6); FROM Asset	Selects the Name column and a calculated column based on the first 6 characters in the Manager column of all records from the Asset table. Note that FoxPro will create its own name for this column unless you use the optional "AS" clause to name it yourself.
4b) SELECT Name, ; LEFT(Manager, 6) AS Mgr_L6; FROM Asset	
5) SELECT COUNT(*) FROM Asset	Determines the count of all records in the Asset table. (See discussion of Statistical SQL Functions below.)

The WHERE Clause

So far our queries have been fairly limited in that they have produced results consisting of the entire table thus, they have not really been queries at all, but just column selectors (or "projection operators" to use the relational jargon). The "WHERE" clause is the first important addition as you build your complete query. Further, it is with the addition of the WHERE clause that you reach the point where you should start testing your query.

The WHERE clause can be very simple or very complex. Basically, the WHERE clause is a FoxPro/SQL expression that can be evaluated as TRUE (.T.) or FALSE (.F.) when the query is run. Only those rows (records) for which this expression evaluates as .T. are further considered. Our syntax has now expanded to:

```

SELECT
    <select_item>
        [AS <column_name>]
        [, [<alias>.] <select_item>
        [AS <column_name>] ...]
FROM <table>
WHERE <filtercondition>

```

There is a wealth of knowledge necessary to be able to construct the most esoteric filter conditions; however, many queries can be quite straightforward using basic relational algebra. e.g.:

6) SELECT Name, Group; FROM Asset; WHERE Group="5"	Selects the Name and Group columns of all records from the Asset table, where the Group field has a value of "5".
--	---

The WHERE clause can be easily expanded using the Boolean operators AND, OR, and NOT; and the relational operators =, ==, >, >=, <, <=, and # (or <>), e.g.:

7) SELECT Name, Group, EndYear; FROM Asset; WHERE Group="3" AND; (EMPTY(EndYear) OR; EndYear>"1999")	Selects the Name and Group columns of all records from the Asset table, where the Group field has a value of "3" and the EndYear field is empty or greater than "1999"..
--	--

Statistical SQL Functions

There are several statistical SQL functions that can be applied to one or more columns in the SELECT statement. Using any one of these completely changes the nature of the result set. This is one of the most frequently misunderstood concepts among new SQL users.

In fact, without use of the "GROUP BY" clause (which is presented later in this discussion), including any statistical function reduces your result set to a single row. Further, including any other (non-statistical) column provides ambiguous (some would say undefined) information, since the output value for those other columns would be randomly based on the last record encountered by the SQL processor, and that processor can access your databases using indexes and other techniques in any sequence it chooses!

The statistical SQL functions are:

```
AVG()  
COUNT()  
MAX()  
MIN()  
SUM()
```

8a) SELECT MAX(EndYear); FROM Asset	Determines the largest value recorded in the EndYear field.
8b) SELECT COUNT(*); FROM Asset; WHERE Group="5"	A very typical "counting" example. Determine how many Group 5 assets are in the database.
8c) SELECT Name, COUNT(*); FROM Asset; WHERE Group="5"	Same as above, but also includes an arbitrary value in the "Name" column. (See previous discussion.)

The GROUP BY Clause

An all-important part of the SQL SELECT syntax is "GROUP BY" clause. This clause specifically causes all rows (records) that survive the WHERE clause filter to be collapsed to a single result row for each unique value of the GROUP BY columns. That is, there will be one row for each group.

The most important concept to understand is:

For queries with no statistical functions, the GROUP BY clause condenses the result from one row per filtered record to just one row per distinct group value; while for queries with one or more statistical functions, the GROUP BY clause expands the result from a single output row having database-wide statistics to one row per distinct group value having group-level statistics.

Obviously, the **GROUP BY** clause is primarily intended for use with the statistical functions. The only purpose of a **GROUP BY** clause, except when combined with statistical functions is to determine the unique values of a given field or combination of fields, e.g.:

9a) SELECT OthOpsStat; FROM Asset; GROUP BY OthOpsStat	Determines each distinct value of the "Other Operational Status" field in the Asset table. (This might be used as <i>ex post</i> analysis for
--	---

	improving on a validation table.)
<pre> 9b) SELECT Area, SubArea; FROM Asset; WHERE Fac_Code="Y12"; GROUP BY 1,2 </pre>	<p>Provide a distinct listing of areas and subareas for the Y-12 plant only (note use of WHERE clause).</p> <p>Note that the GROUP BY clause can employ <i>numeric</i> column designators in place of field names. This is <u>not recommended</u> as a general practice, because changing the SELECT field list could alter the grouping inadvertently.</p>
<pre> 9c) SELECT Area, SubArea; FROM Asset; WHERE Fac_Code="Y12"; GROUP BY 1,2 </pre>	<p>Provide a distinct listing of areas and subareas for the Y-12 plant only (note use of WHERE clause).</p> <p>Note that the GROUP BY clause can employ <i>numeric</i> column designators in place of field names. This is <u>not recommended</u> as a general practice, because changing the SELECT field list could alter the grouping inadvertently.</p>

The **GROUP BY** clause really becomes valuable when used with the statistical functions. Most commonly, the SELECT column list includes the grouping identifiers and one statistical function, e.g.:

<pre>10a)SELECT OwnerSO,Group,COUNT(*) ; FROM Asset ; WHERE Type=1 ; GROUP BY OwnerSO, Group</pre>	<p>Determine a tabulation of Type 1 records from the Asset table based on the combination of the Group and OwnerSO fields.</p> <p>NOTE: This 3-column query is frequently referred to as a “cross-tabulation” or “cross-tab”, and is often displayed in a table with the two group field’s distinct values comprising the row and column headings. FoxPro’s GENXTAB program can be used to help automate this process.</p> <p>To operate GENXTAB, create a 3-field cursor or table (such as the example query), and simply type DO GENXTAB from the Command Window.</p>
<pre>10b) SELECT OwnerSO, COUNT(*) ; FROM Asset ; WHERE Type=1 ; GROUP BY OwnerSO</pre>	<p>Perform a higher level grouping, resulting in totals based on one field only.</p> <p>NOTE: This 2-column query can be thought of as a “histogram” query, in that it produces group totals for a single field’s values ideal for producing a histogram.</p>

Note the examples above. Specifically, the SELECT statement can’t be told to produce interspersed subtotals. Thus query 10b) must be run even if query 10a) was also run unless you use a program or report to operate further on the results of query 10a).

CAUTION

Field names that are identical to “reserved” FoxPro words can create major problems and should generally be avoided.

In the above examples, you cannot list the field “Group” first, because FoxPro confuses that with a **GROUP BY** clause and produces a Syntax Error. You can work around this problem by prepending the field name with the table name (i.e.; **SELECT Asset.Group**, OwnerSO, COUNT(*) FROM Asset)

The HAVING Clause

The HAVING clause only applies to queries that have a GROUP BY clause as well. The condition specified in the HAVING clause is applied to the grouped output and only those groups meeting the condition specified are included in the final result. The distinction from the WHERE clause is that the WHERE conditions are applied to each individual database record, while the HAVING conditions are tests that can be applied only in aggregate, after the data has been filtered and grouped. An example:

<pre>11) SELECT Fac_Code, COUNT(*); FROM Asset; WHERE Group="3"; GROUP BY Fac_Code HAVING COUNT(*)>3</pre>	<p>Determines those facilities that have more than 3 Group 3 assets. Notice how the WHERE clause limits the domain to those assets of interest (Group 3), while the HAVING clause waits until the Group 3 assets have been counted up by Facility and then filters out any Facilities with the specified counts.</p> <p>NOTE: If the HAVING clause is included in a query with no GROUP BY it behaves as though it were a WHERE clause. Nevertheless, this is a confusing practice and should be avoided.</p>
---	---

The ORDER BY Clause

None of the clauses introduced so far have any necessary effect on the sequence in which the final result set is displayed. The ORDER BY clause can be used to accomplish this task. The ORDER BY clause often includes some or all of the same fields used in the SELECT list (or the GROUP BY clause, if there is one).

<pre>12a)SELECT OwnerSO, Group,COUNT(*); FROM Asset; WHERE Type=1; GROUP BY OwnerSO, Group,; ORDER BY OwnerSO, Group,</pre>	<p>Determine a tabulation of Type 1 records from the Asset table based on the combination of the Group and OwnerSO fields <u>and</u> ensure the output is ordered by OwnerSO and then by Group.</p>
---	---

The ORDER BY clause has no effect on the result set other than physical display order.

Where the Results Go

Normally, the results of a SELECT statement are provided in a "cursor" (FoxPro's term for a pseudo-table used to display temporary results), which is then BROWSE'd on screen. There are many alternatives to this, including creating a physical DBF file or sending the results to the printer or Microsoft Excel. These options are accomplished via the "TO" and "INTO" clauses, which are discussed in the next lesson.

Relational Query By Example (RQBE)

RQBE is a FoxPro "power tool" that helps to automate the construction of simpler SQL SELECT statements. RQBE provides a visual interface and then constructs the equivalent SELECT statement in the background. Fortunately, FoxPro allows you to see the fruits of its labors via the "See SQL" push button. Even better, saving an RQBE session creates an .QPR file, which is nothing but the SELECT statement itself. This statement can be copied into your programs, or just to the Command Window, where you can further modify it.

RQBE can also help to teach SQL to you just provide the attributes of the desired query visually, and make frequent use of the "See SQL" function to observe how FoxPro has implemented your request.

To initiate an RQBE session, simply use the File New dialog and choose "Query" as the File type. As an alternative, you can use MODIFY QUERY from the Command Window. You should note that RQBE cannot be used to construct many types of advanced queries that are covered in the next lesson.

Answer to This Week's Car Talk "Puzzler"

```
SELECT City, COUNT(*) AS Instances ;  
FROM Cities ;  
GROUP BY City ;  
INTO CURSOR NameCount
```

```
SELECT City, MAX(Instances) ;  
FROM NameCount
```

The Basics of Querying in FoxPro with SQL SELECT

(Lesson 2 Multiple Table Queries)

The SELECT syntax that has been covered through Lesson 1 is:

```
SELECT [<alias>.] <select_item>
      [AS <column_name>]
      [, [<alias>.]<select_item>
      [AS <column_name>] ...]
FROM <table>
[WHERE <filtercondition>
[AND \ OR <filtercondition> ...]]
[GROUP BY <groupcolumn>
[, <groupcolumn> ...] ]
[HAVING <filtercondition>]
[ORDER BY <order_item>
[, <order_item>]]
```

Attaining Data from Additional Tables

The queries in Lesson 1 were similar in that the results were all derived from a single database table (ASSET.DBF). Frequently you will need answers that must be derived from two or more tables. This occurs in two situations:

- The primary table uses a “code” field that provides a “lookup reference” into another table, and information from that “reference table” is needed to answer the query.
- There is a database relationship between two or more tables (one-to-many or many-to-many) and the information in the tables must be combined to answer the query.

Without FoxPro’s SELECT statement it can take significant programming to accomplish these tasks. Fortunately, some minor additions to our SELECT syntax solves the problem:

1. Extend the FROM clause to include all tables to be combined.
2. Extend the WHERE clause to include “join conditions”, which are AND-combined with any other filter conditions. Join conditions indicate to FoxPro what fields are to be used to determine which records from the second table are related to any given record in the first table. This is a critical step. FoxPro makes no assumption about the relationship between files. For example, just because two tables each have field names of “Fac_Code”, FoxPro does not guess that this field should be used for the join condition.

Also note that failure to include a join condition produces the dread “Cartesian Product” in which there is one row in the result set for each possible combination of rows in the original tables. Thus two 10,000 record tables joined without condition would produce a 100,000,000 row result set! (Usually, it produces a “Disk Full” error instead.)

3. If necessary, clarify all field references in all clauses to indicate which table each one comes from. Unless you do this, any fields in the query that exist by the same name in more than one table will create a runtime error.

An example:

<pre>SELECT Asset.Fac_Code,Asset.AssetNo,Facility.Name; FROM Asset, Facility; WHERE Asset.Fac_Code=Facility.Fac_Code</pre>	<p>In addition to the main identifiers, show the Facility's full name.</p>
--	--

Frequently, the main purpose of the join is to filter or group based on a field in the second table. All of the capabilities presented in Lesson 1 apply to two-table joins. Here's a larger example:

<pre>SELECT Facility.Ops_Office,Asset.OwnerSO,; Asset.Group, COUNT(*); FROM Asset, Facility; WHERE Asset.Fac_Code = ; Facility.Fac_Code; GROUP BY 1,2,3</pre>	<p>Produce a three-tiered tabulation based on, first, the Operations Office (which must be derived from the Facility table), and then on two familiar fields.</p>
---	---

The Basics of Querying in FoxPro with SQL SELECT

(Lesson 3 Environmental Considerations)

Several aspects of the current FoxPro environment may affect the speed or actual results of your queries. The following is probably incomplete:

Environment Factor	Effects on Queries
SET DELETED	<p>Results: If DELETED is set to OFF, all deleted records are included in the result set. This is generally <u>not</u> the desired result. Unfortunately, this is also FoxPro's startup default. (See separate discussion below for how to change this.)</p> <p>Speed: If DELETED is set to ON, and your tables do not have an index TAG based on the DELETED() function, your queries may run significantly slower. Remember that with DELETED ON, FoxPro is, in effect, performing an implicit "FOR NOT DELETED()" filtration on all database operations.</p>
SET EXACT	<p>Results: If EXACT is set ON, the results of the "=" and other operators can yield different results than with EXACT OFF. I recommend that EXACT be set OFF at all times. (See separate discussion below for how to change this.)</p> <p>Speed: N/A</p>
SET ANSI	<p>Results: Must be set to OFF (the default) or erroneous results may be attained.</p> <p>Speed: N/A</p>
USE Statements	<p>Results: Whether tables are open or not has no effect whatsoever on the results of a SELECT statement. (That is, there is no need to USE the tables first.)</p> <p>Speed: Speed can be significantly affected, particularly if a table is going to be queried repeatedly. DOS is very slow at things like opening and closing files. You should open them and leave them open.</p> <p>(Note that if the SELECT statement requires certain previously-unopened files to be opened, they are not closed when the query is completed. Failure to manage your environment can lead to eventually exceeding the FILES= limit in your CONFIG.SYS.)</p>

SET FILTER (or equivalent)	<p>Results: Setting a FILTER on an open table prior to running a SELECT has <u>no effect</u> whatsoever on your query result. FoxPro ignores all active indexes and filters (other than the DELETED setting) when executing a query. Any desired filtering must be accomplished via the WHERE clause of the SELECT statement itself.</p> <p>Speed: N/A</p>
SET INDEX TO or SET ORDER TO	<p>Results: Ordering records via an open index prior to running a SELECT has <u>no effect</u> whatsoever on your query result. FoxPro ignores all active indexes when executing a query. Any desired ordering must be accomplished via the ORDER BY clause of the SELECT statement itself.</p> <p>Speed: The <u>existence</u> of indexes, on the other hand, can make a dramatic impact on the speed of the query to the extent those indexes are based on fields specified in the WHERE clause (and to a lesser degree the GROUP BY or ORDER BY clauses).</p> <p>FoxPro employs a technology known as Rushmore to speed the results of database set operations (such as SQL SELECT). This technology is invoked when indexes, which can be efficiently cached into RAM, contain all or part of the information necessary to resolve a query. For SELECT statements, this technology is deployed whenever the indexes exist in the production index file (.CDX) <u>or</u> if other open indexes are available when the command is executed. The Rushmore technology can take advantage of several indexes simultaneously to speed complex queries.</p>
SET OPTIMIZE	<p>Results: This setting has no impact on results.</p> <p>Speed: If set to OFF, the Rushmore technology is disabled. There is generally no need to do this for SELECT statements. Certain other operations can be enhanced by temporarily turning Rushmore off. One example is GO TOP or GO BOTTOM when a FILTER and an INDEX are both set.</p>

<p>Using = = instead of = in your WHERE clauses.</p>	<p>This is a messy one, with potential effects on both results and speed. The = = operator means “precisely equals”, and for character expressions evaluates to .T. only when both operands have the same length as well as the same value. The regular = operator behaves in different ways based on the setting of SET EXACT.</p> <p>With EXACT ON = = and = are identical. With EXACT OFF, a longer left operand is “equal” to a shorter right operand if the left matches the right up to the point where the right operand terminates (e.g., “SMITHSEN” = “SMITH” would evaluate to .T., but not vice versa). Note that spaces are significant, so when comparing two <u>fields</u> with the same length, the = = operator is never necessary.</p> <p>Results: Because I recommend that EXACT always be set to OFF, these two operators provide two separate behaviors for the result set that you desire.</p> <p>Speed: It is not well known, but Rushmore is disabled when the = = operator is used. Thus, if you don’t need = =, use a plain =. (Similarly, don’t select the “Exactly Equals” choice in RQBE.)</p>
--	--

Specifying FoxPro’s Startup Defaults

If you are (properly) concerned about FoxPro’s setting of DELETED, EXACT, and ANSI; you can take action to ensure they are correct. There are two solutions to this. The first would be to remember to type the following lines in the Command Window every time you start FoxPro:

```
SET DELETED ON
SET EXACT OFF
SET ANSI ON
SET UNIQUE OFF (another dangerous one I’m slipping in)
```

This isn’t a desirable task to perform every time, and sooner or later you would forget. Fortunately, all of these can be set via a FoxPro configuration file, which FoxPro can be told to load automatically upon startup. This file is typically named CONFIG.FP and placed in the FPD26 directory (or wherever FoxPro is installed). The file can be created from DOS (EDIT CONFIG.FP) or FoxPro (MODIFY COMMAND CONFIG.FP), and would appear as follows:

```
DELETED=ON
EXACT=OFF
ANSI=OFF
UNIQUE=OFF
```

FoxPro uses the following priorities to decide which configuration file to load, if any, upon startup:

1. If a compiled application is run and a (read-only) CONFIG.FP file has been bound into the .APP or .EXE file, that configuration is loaded; otherwise,
2. if a configuration file is specified from the DOS command line via the “-c” startup switch, that configuration file is loaded; otherwise,

(Example: `FOXPROX -cC:\FPD26\MY_PREFS.FP`)

3. if a DOS environment variable `FOXPROCFG` exists, FoxPro loads the configuration file specified by that variable; otherwise,

(Example: `SET FOXPROCFG =c:\fpd26\my_prefs.fp`)

4. if a file named `CONFIG.FP` exists in the current DOS directory when FoxPro is started, FoxPro loads that configuration file; otherwise,
5. if a file named `CONFIG.FP` exists in the FoxPro program directory, FoxPro loads that configuration file; otherwise,
6. no configuration file is loaded.