Introduction to Structured Query Language

Version 3.53

By: Jim Hoffman

jhoffman@one.net.

Introduction

This is a tutorial of the* Structured Query Language *(also known as* SQL*) and is a pioneering effort on the World Wide Web, as this is the first comprehensive SQL tutorial available on the Internet. SQL allows users to access data in relational database management systems, such as Oracle, Sybase, Informix, Microsoft SQL Server, Access, and others, by allowing users to describe the data the user wishes to see. SQL also allows users to define the data in a database, and manipulate that data. This document will describe how to use SQL, and give examples. The SQL used in this document is "ANSI", or standard SQL, and no SQL features of specific database management systems will be discussed until the "Nonstandard SQL" section.

Comments or suggestions? Mail me at jhoffman@one.net.

Copyright 1996-1998, James Hoffman. This document can be used for free by any Internet user, but cannot be included in another document, published in any other form, or mass produced in any way.

Last updated: 1-10-1998; miscellaneous tasks.

Table of Contents

Section 1.1 Basics of the SELECT Statement

In a relational database, data is stored in tables. An example table would relate Social Security Number, Name, and Address:

EmployeeAddressTable					
SSN	**FirstName**	**LastName**	**Address**	**City**	**State**
512687458	Joe	Smith	83 First Street	Howard	Ohio
758420012	Mary	Scott	842 Vine Ave.	Losantiville	Ohio
102254896	Sam	Jones	33 Elm St.	Paris	New York
876512563	Sarah	Ackerman	440 U.S. 110	Upton	Michigan

Now, let's say you want to see the address of each employee. Use the SELECT statement, like so:

```
SELECT FirstName, LastName, Address, City, State
FROM EmployeeAddressTable;
```

The following is the results of your *query* of the database:

First Name	Last Name	Address	City	State
Joe	Smith	83 First Street	Howard	Ohio
Mary	Scott	842 Vine Ave.	Losantiville	Ohio
Sam	Jones	33 Elm St.	Paris	New York
Sarah	Ackerman	440 U.S. 110	Upton	Michigan

To explain what you just did, you asked for the all of data in the EmployeeAddressTable, and specifically, you asked for the *columns* called FirstName, LastName, Address, City, and State. Note that column names and table names do not have spaces...they must be typed as one word; and that the statement ends with a semicolon (;). The general form for a SELECT statement, retrieving all of the *rows* in the table is:

```
SELECT ColumnName, ColumnName, ...
FROM TableName;
```

To get all columns of a table without typing all column names, use:

```
SELECT * FROM TableName;
```

Each database management system (DBMS) and database software has different methods for logging in to the database and entering SQL commands; see the local computer "guru" to help you get onto the system, so that you can use SQL.

Section 1.2 Conditional Selection

To further discuss the SELECT statement, let's look at a new example table (for hypothetical purposes only):

EmployeeStatisticsTable

EmployeeIDNo	Salary	Benefits	Position
010	75000	15000	Manager
105	65000	15000	Manager
152	60000	15000	Manager
215	60000	12500	Manager
244	50000	12000	Staff
300	45000	10000	Staff
335	40000	10000	Staff
400	32000	7500	Entry-Level
441	28000	7500	Entry-Level

Section 1.3 Relational Operators

There are six Relational Operators in SQL, and after introducing them, we'll see how they're used:

=	Equal
<> or != (see manual)	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To

The *WHERE* clause is used to specify that only certain rows of the table are displayed, based on the

criteria described in that *WHERE clause*. It is most easily understood by looking at a couple of examples.

If you wanted to see the EMPLOYEEIDNO's of those making at or over $50,000, use the following:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY >= 50000;
```

Notice that the >= (greater than or equal to) sign is used, as we wanted to see those who made greater than $50,000, or equal to $50,000, listed together. This displays:

```
EMPLOYEEIDNO
------------
010
105
152
215
244
```

The *WHERE* description, SALARY >= 50000, is known as a *condition.* The same can be done for text columns:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This displays the ID Numbers of all Managers. Generally, with text columns, stick to equal to or not equal to, and make sure that any text that appears in the statement is surrounded by single quotes (').

Section 1.4 More Complex Conditions: Compound Conditions

The *AND* operator joins two or more conditions, and displays a row only if that row's data satisfies **ALL** conditions listed (i.e. all conditions hold true). For example, to display all staff making over $40,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY > 40000 AND POSITION = 'Staff';
```

The *OR* operator joins two or more conditions, but returns a row if **ANY** of the conditions listed hold true. To see all those who make less than $40,000 or have less than $10,000 in benefits, listed together, use the following query:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY < 40000 OR BENEFITS < 10000;
```

AND & OR can be combined, for example:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND SALARY > 60000 OR BENEFITS > 12000;
```

First, SQL finds the rows where the salary is greater than $60,000 and the position column is equal to Manager, then taking this new list of rows, SQL then sees if any of these rows satisfies the previous AND condition or the condition that the Benefits column is greater then $12,000. Subsequently, SQL only displays this second new list of rows, keeping in mind that anyone with Benefits over $12,000 will be included as the OR operator includes a row if either resulting condition is True. Also note that the AND operation is done first.

To generalize this process, SQL performs the AND operation(s) to determine the rows where the AND operation(s) hold true (remember: all of the conditions are true), then these results are used to compare with the OR conditions, and only display those remaining rows where the conditions joined by the OR operator hold true.

To perform OR's before AND's, like if you wanted to see a list of employees making a large salary (>$50,000) or have a large benefit package (>$10,000), and that happen to be a manager, use parentheses:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND (SALARY > 50000 OR BENEFIT > 10000);
```

Section 1.5 *IN & BETWEEN*

An easier method of using compound conditions uses *IN* or *BETWEEN*. For example, if you wanted to list all managers and staff:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION IN ('Manager', 'Staff');
```

or to list those making greater than or equal to $30,000, but less than or equal to $50,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY BETWEEN 30000 AND 50000;
```

To list everyone not in this range, try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY NOT BETWEEN 30000 AND 50000;
```

Similarly, NOT IN lists all rows excluded from the *IN* list.

Section 1.6 *Using LIKE*

Look at the EmployeeStatisticsTable, and say you wanted to see all people whose last names started with "L"; try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEEADDRESSTABLE
WHERE LASTNAME LIKE 'L%';
```

The percent sign (%) is used to represent any possible character (number, letter, or punctuation) or set of characters that might appear after the "L". To find those people with LastName's ending in "L", use '%L', or if you wanted the "L" in the middle of the word, try '%L%'. The '%' can be used for any characters in the same position relative to the given characters. NOT LIKE displays rows not fitting the given description. Other possiblities of using LIKE, or any of these discussed conditionals, are available, though it depends on what DBMS you are using; as usual, consult a manual or your system manager or administrator for the available features on your system, or just to make sure that what you are trying to do is available and allowed. This disclaimer holds for the features of SQL that will be discussed below. This section is just to give you an idea of the possibilities of queries that can be written in SQL.

Section 2.1 Joins

In this section, we will only discuss *inner* joins, and *equijoins*, as in general, they are the most useful. For more information, try the SQL links at the bottom of the page.

Good database design suggests that each table lists data only about a single *entity*, and detailed information can be obtained in a relational database, by using additional tables, and by using a *join*.

First, take a look at these example tables:

AntiqueOwners

OwnerID	**OwnerLastName**	**OwnerFirstName**
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Orders

OwnerID	**ItemDesired**
02	Table
02	Desk
21	Chair
15	Mirror

Antiques

SellerID	**BuyerID**	**Item**

01	50	Bed
02	15	Table
15	02	Chair
21	50	Mirror
50	01	Desk
01	21	Cabinet
02	21	Coffee Table
15	50	Chair
01	15	Jewelry Box
02	21	Pottery
21	02	Bookcase
50	01	Plant Stand

Section 2.2 *Keys*

First, let's discuss the concept of *keys*. A *primary key* is a column or set of columns that uniquely identifies the rest of the data in any given row. For example, in the AntiqueOwners table, the OwnerID column uniquely identifies that row. This means two things: no two rows can have the same OwnerID, and, even if two owners have the same first and last names, the OwnerID column ensures that the two owners will not be confused with each other, because the unique OwnerID column will be used throughout the database to track the owners, rather than the names.

A *foreign key* is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key. In DBMS-speak, this correspondence is known as *referential integrity*. For example, in the Antiques table, both the BuyerID and SellerID are foreign keys to the primary key of the AntiqueOwners table (OwnerID; for purposes of argument, one has to be an Antique Owner before one can buy or sell any items), as, in both tables, the ID rows are used to identify the owners or buyers and sellers, and that the OwnerID is the primary key of the AntiqueOwners table. In other words, all of this "ID" data is used to refer to the owners, buyers, or sellers of antiques, themselves, without having to use the actual names.

Section 2.3 Performing a Join

The purpose of these *keys* is so that data can be related across tables, without having to repeat data in every table--this is the power of relational databases. For example, you can find the names of those who bought a chair without having to list the full name of the buyer in the Antiques table...you can get the name by relating those who bought a chair with the names in the AntiqueOwners table through the use of the OwnerID, which *relates* the data in the two tables. To find the names of those who bought a chair, use the following query:

```
SELECT OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE BUYERID = OWNERID AND ITEM = 'Chair';
```

Note the following about this query...notice that both tables involved in the relation are listed in the FROM clause of the statement. In the WHERE clause, first notice that the ITEM = 'Chair' part restricts the listing to those who have bought (and in this example, thereby owns) a chair. Secondly, notice how the ID columns are related from one table to the next by use of the BUYERID = OWNERID clause. Only where ID's match across tables and the item purchased is a chair (because of the AND), will the names from the AntiqueOwners table be listed. Because the joining condition used an equal sign, this join is called an *equijoin*. The result of this query is two names: Smith, Bob & Fowler, Sam.

Dot notation refers to prefixing the table names to column names, to avoid ambiguity, as such:

```
SELECT ANTIQUEOWNERS.OWNERLASTNAME, ANTIQUEOWNERS.OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE ANTIQUES.BUYERID = ANTIQUEOWNERS.OWNERID AND ANTIQUES.ITEM = 'Chair';
```

As the column names are different in each table, however, this wasn't necessary.

Section 2.4 DISTINCT and Eliminating Duplicates

Let's say that you want to list the ID and names of **only** those people who have sold an antique. Obviously, you want a list where each seller is only listed once--you don't want to know how many antiques a person sold, just the fact that this person sold one (for counts, see the Aggregate Function section below). This means that you will need to tell SQL to eliminate duplicate sales rows, and just list each person only once. To do this, use the *DISTINCT* keyword.

First, we will need an equijoin to the AntiqueOwners table to get the detail data of the person's LastName and FirstName. However, keep in mind that since the SellerID column in the Antiques table is a foreign key to the AntiqueOwners table, a seller will only be listed if there is a row in the AntiqueOwners table listing the ID and names. We also want to eliminate multiple occurences of the SellerID in our listing, so we use *DISTINCT* **on the column where the repeats may occur.**

To throw in one more twist, we will also want the list alphabetized by LastName, then by FirstName (on a LastName tie), then by OwnerID (on a LastName and FirstName tie). Thus, we will use the *ORDER BY* clause:

```
SELECT DISTINCT SELLERID, OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUES, ANTIQUEOWNERS
WHERE SELLERID = OWNERID
ORDER BY OWNERLASTNAME, OWNERFIRSTNAME, OWNERID;
```

In this example, since everyone has sold an item, we will get a listing of all of the owners, in alphabetical order by last name. For future reference (and in case anyone asks), this type of join is considered to be in the category of *inner joins*.

Section 2.5 Aliases & In/Subqueries

In this section, we will talk about *Aliases*, *In* and the use of subqueries, and how these can be used in a 3-table example. First, look at this query which prints the last name of those owners who have placed an order and what the order is, only listing those orders which can be filled (that is, there is a buyer who owns that ordered item):

```
SELECT OWN.OWNERLASTNAME Last Name, ORD.ITEMDESIRED Item Ordered
FROM ORDERS ORD, ANTIQUEOWNERS OWN
WHERE ORD.OWNERID = OWN.OWNERID
AND ORD.ITEMDESIRED IN

   (SELECT ITEM
   FROM ANTIQUES);
```

This gives:

```
Last Name Item Ordered
--------- ------------
Smith     Table
Smith     Desk
Akins     Chair
Lawson    Mirror
```

There are several things to note about this query:

1. First, the "Last Name" and "Item Ordered" in the Select lines gives the headers on the report.
2. The OWN & ORD are aliases; these are new names for the two tables listed in the FROM clause that are used as prefixes for all dot notations of column names in the query (see above). This eliminates ambiguity, especially in the equijoin WHERE clause where both tables have the column named OwnerID, and the dot notation tells SQL that we are talking about two different OwnerID's from the two different tables.
3. Note that the Orders table is listed first in the FROM clause; this makes sure listing is done off of that table, and the AntiqueOwners table is only used for the detail information (Last Name).
4. Most importantly, the AND in the WHERE clause forces the In Subquery to be invoked ("=

ANY" or "= SOME" are two equivalent uses of IN). What this does is, the subquery is performed, returning all of the Items owned from the Antiques table, as there is no WHERE clause. Then, for a row from the Orders table to be listed, the ItemDesired must be in that returned list of Items owned from the Antiques table, thus listing an item only if the order can be filled from another owner. You can think of it this way: the subquery returns a *set* of Items from which each ItemDesired in the Orders table is compared; the In condition is true only if the ItemDesired is in that returned set from the Antiques table.

5. Also notice, that in this case, that there happened to be an antique available for each one desired...obviously, that won't always be the case. In addition, notice that when the IN, "= ANY", or "= SOME" is used, that these keywords refer to any possible row matches, not column matches...that is, you cannot put multiple columns in the subquery Select clause, in an attempt to match the column in the outer Where clause to one of multiple possible column values in the subquery; only one column can be listed in the subquery, and the possible match comes from multiple *row* values in that *one* column, not vice-versa.

Whew! That's enough on the topic of complex SELECT queries for now. Now on to other SQL statements.

Section 3.1 Miscellaneous SQL Statements: Aggregate Functions

I will discuss five important *aggregate functions*: SUM, AVG, MAX, MIN, and COUNT. They are called aggregate functions because they summarize the results of a query, rather than listing all of the rows.

- SUM () gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.
- AVG () gives the average of the given column.
- MAX () gives the largest figure in the given column.
- MIN () gives the smallest figure in the given column.
- COUNT(*) gives the number of rows satisfying the conditions.

Looking at the tables at the top of the document, let's look at three examples:

```
SELECT SUM(SALARY), AVG(SALARY)
FROM EMPLOYEESTATISTICSTABLE;
```

This query shows the total of all salaries in the table, and the average salary of all of the entries in the table.

```
SELECT MIN(BENEFITS)
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This query gives the smallest figure of the Benefits column, of the employees who are Managers, which is 12500.

```
SELECT COUNT(*)
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Staff';
```

This query tells you how many employees have Staff status (3).

Section 3.2 Miscellaneous SQL Statements: Views

In SQL, you might (check your DBA) have access to create views for yourself. What a view does is to allow you to assign the results of a query to a new, personal table, that you can use in other queries, where this new table is given the view name in your FROM clause. When you access a view, the query that is defined in your view creation statement is performed (generally), and the results of that query look just like another table in the query that you wrote invoking the view. For example, to create a view:

```
CREATE VIEW ANTVIEW AS SELECT ITEMDESIRED FROM ORDERS;
```

Now, write a query using this view as a table, where the table is just a listing of all Items Desired from the Orders table:

```
SELECT SELLERID
FROM ANTIQUES, ANTVIEW
WHERE ITEMDESIRED = ITEM;
```

This query shows all SellerID's from the Antiques table where the Item in that table happens to appear in the Antview view, which is just all of the Items Desired in the Orders table. The listing is generated by going through the Antique Items one-by-one until there's a match with the Antview view. Views can be used to restrict database access, as well as, in this case, simplify a complex query.

Section 3.3 *Miscellaneous SQL Statements: Creating New Tables*

All tables within a database must be created at some point in time...let's see how we would create the Orders table:

```
CREATE TABLE ORDERS
(OWNERID INTEGER NOT NULL,
ITEMDESIRED CHAR(40) NOT NULL);
```

This statement gives the table name and tells the DBMS about each column in the table. ***Please note*** that this statement uses generic data types, and that the data types might be different, depending on what DBMS you are using. As usual, check local listings. Some common generic data types are:

- Char(x) - A column of characters, where x is a number designating the maximum number of characters allowed (maximum length) in the column.
- Integer - A column of whole numbers, positive or negative.
- Decimal(x, y) - A column of decimal numbers, where x is the maximum length in digits of the decimal numbers in this column, and y is the maximum number of digits allowed after the decimal point. The maximum (4,2) number would be 99.99.
- Date - A date column in a DBMS-specific format.
- Logical - A column that can hold only two values: TRUE or FALSE.

One other note, the NOT NULL means that the column must have a value in each row. If NULL was used, that column may be left empty in a given row.

Section 3.4 Miscellaneous SQL Statements: Altering Tables

Let's add a column to the Antiques table to allow the entry of the price of a given Item:

```
ALTER TABLE ANTIQUES ADD (PRICE DECIMAL(8,2) NULL);
```

The data for this new column can be updated or inserted as shown later.

Section 3.5 Miscellaneous SQL Statements: Adding Data

To insert rows into a table, do the following:

```
INSERT INTO ANTIQUES VALUES (21, 01, 'Ottoman', 200.00);
```

This inserts the data into the table, as a new row, column-by-column, in the pre-defined order. Instead, let's change the order and leave Price blank:

```
INSERT INTO ANTIQUES (BUYERID, SELLERID, ITEM)
VALUES (01, 21, 'Ottoman');
```

Section 3.6 Miscellaneous SQL Statements: Deleting Data

Let's delete this new row back out of the database:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman';
```

But if there is another row that contains 'Ottoman', that row will be deleted also. Let's delete all rows (one, in this case) that contain the specific data we added before:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman' AND BUYERID = 01 AND SELLERID = 21;
```

Section 3.7 Miscellaneous SQL Statements: Updating Data

Let's update a Price into a row that doesn't have a price listed yet:

```
UPDATE ANTIQUES SET PRICE = 500.00 WHERE ITEM = 'Chair';
```

This sets all Chair's Prices to 500.00. As shown above, more WHERE conditionals, using AND, must be used to limit the updating to more specific rows. Also, additional columns may be set by separating equal statements with commas.

Section 4.1 Miscellaneous Topics: *Indexes*

Indexes allow a DBMS to access data quicker (*please note:* this feature is nonstandard/not available on all systems). The system creates this internal data structure (the index) which causes selection of rows, when the selection is based on indexed columns, to occur faster. This index tells the DBMS where a certain row is in the table given an indexed-column value, much like a book index tells you what page a given word appears. Let's create an index for the OwnerID in the AntiqueOwners column:

```
CREATE INDEX OID_IDX ON ANTIQUEOWNERS (OWNERID);
```

Now on the names:

```
CREATE INDEX NAME_IDX ON ANTIQUEOWNERS (OWNERLASTNAME, OWNERFIRSTNAME);
```

To get rid of an index, drop it:

```
DROP INDEX OID_IDX;
```

By the way, you can also "drop" a table, as well (careful!--that means that your table is deleted). In the second example, the index is kept on the two columns, aggregated together--strange behavior might occur in this situation...check the manual before performing such an operation.

Some DBMS's do not enforce primary keys; in other words, the uniqueness of a column is not enforced automatically. What that means is, if, for example, I tried to insert another row into the AntiqueOwners table with an OwnerID of 02, some systems will allow me to do that, even though, we do not, as that column is supposed to be unique to that table (every row value is supposed to be different). One way to get around that is to create a unique index on the column that we want to be a primary key, to force the system to enforce prohibition of duplicates:

```
CREATE UNIQUE INDEX OID_IDX ON ANTIQUEOWNERS (OWNERID);
```

Section 4.2 Miscellaneous Topics: *GROUP BY & HAVING*

One special use of GROUP BY is to associate an aggregate function (especially COUNT; counting the number of rows in each group) with groups of rows. First, assume that the Antiques table has the Price column, and each row has a value for that column. We want to see the price of the most expensive item bought by each owner. We have to tell SQL to *group* each owner's purchases, and tell us the maximum purchase price:

```
SELECT BUYERID, MAX(PRICE)
FROM ANTIQUES
GROUP BY BUYERID;
```

Now, say we only want to see the maximum purchase price if the purchase is over $1000, so we use the HAVING clause:

```
SELECT BUYERID, MAX(PRICE)
FROM ANTIQUES
GROUP BY BUYERID
HAVING PRICE > 1000;
```

Section 4.3 Miscellaneous Topics: *More Subqueries*

Another common usage of subqueries involves the use of operators to allow a Where condition to include the Select output of a subquery. First, list the buyers who purchased an expensive item (the Price of the item is $100 greater than the average price of all items purchased):

```
SELECT BUYERID
FROM ANTIQUES
WHERE PRICE >

   (SELECT AVG(PRICE) + 100
   FROM ANTIQUES);
```

The subquery calculates the average Price, plus $100, and using that figure, an OwnerID is printed for every item costing over that figure. One could use DISTINCT OWNERID, to eliminate duplicates.

List the Last Names of those in the AntiqueOwners table, ONLY if they have bought an item:

```
SELECT OWNERLASTNAME
FROM ANTIQUEOWNERS
WHERE OWNERID IN

   (SELECT DISTINCT BUYERID
   FROM ANTIQUES);
```

The subquery returns a list of buyers, and the Last Name is printed for an Antique Owner if and only if the Owner's ID appears in the subquery list (sometimes called a *candidate list*). *Note:* on some DBMS's, equals can be used instead of IN, but for clarity's sake, since a set is returned from the subquery, IN is the better choice.

For an Update example, we know that the gentleman who bought the bookcase has the wrong First Name in the database...it should be John:

```
UPDATE ANTIQUEOWNERS
SET OWNERFIRSTNAME = 'John'
WHERE OWNERID =

   (SELECT BUYERID
   FROM ANTIQUES
   WHERE ITEM = 'Bookcase');
```

First, the subquery finds the BuyerID for the person(s) who bought the Bookcase, then the outer query updates his First Name.

Remember this rule about subqueries: when you have a subquery as part of a WHERE condition, the Select clause in the subquery must have columns that match in number and type to those in the Where clause of the outer query. In other words, if you have "**`WHERE ColumnName = (SELECT...);`**", the Select must have only one column in it, to match the ColumnName in the outer Where clause, *and* they must match in type (both being integers, both being character strings, etc.).

Section 4.4 Miscellaneous Topics: *EXISTS & ALL*

EXISTS uses a subquery as a condition, where the condition is True if the subquery returns any rows, and False if the subquery does not return any rows; this is a nonintuitive feature with few unique uses. However, if a prospective customer wanted to see the list of Owners only if the shop dealt in Chairs, try:

```
SELECT OWNERFIRSTNAME, OWNERLASTNAME
FROM ANTIQUEOWNERS
WHERE EXISTS

   (SELECT *
   FROM ANTIQUES
   WHERE ITEM = 'Chair');
```

If there are any Chairs in the Antiques column, the subquery would return a row or rows, making the EXISTS clause true, causing SQL to list the Antique Owners. If there had been no Chairs, no rows would have been returned by the outside query.

ALL is another unusual feature, as ALL queries can usually be done with different, and possibly simpler methods; let's take a look at an example query:

```
SELECT BUYERID, ITEM
FROM ANTIQUES
WHERE PRICE >= ALL

   (SELECT PRICE
   FROM ANTIQUES);
```

This will return the largest priced item (or more than one item if there is a tie), and its buyer. The subquery returns a list of all Prices in the Antiques table, and the outer query goes through each row of the Antiques table, and if its Price is greater than or equal to every (or ALL) Prices in the list, it is listed, giving the highest priced Item. The reason ">=" must be used is that the highest priced item will be equal to the highest price on the list, because this Item is in the Price list.

Section 4.5 Miscellaneous Topics: *UNION & Outer Joins*

There are occasions where you might want to see the results of multiple queries together, combining their output; use UNION. To merge the output of the following two queries, displaying the ID's of all

Buyers, plus all those who have an Order placed:

```
SELECT BUYERID
FROM ANTIQUES
UNION
SELECT OWNERID
FROM ORDERS;
```

Notice that SQL requires that the Select list (of columns) must match, column-by-column, in data type. In this case BuyerID and OwnerID are of the same data type (integer). Also notice that SQL does automatic duplicate elimination when using UNION (as if they were two "sets"); in single queries, you have to use DISTINCT.

The *outer join* is used when a join query is "united" with the rows not included in the join, and are especially useful if constant text "flags" are included. First, look at the query:

```
SELECT OWNERID, 'is in both Orders & Antiques'
FROM ORDERS, ANTIQUES
WHERE OWNERID = BUYERID
UNION
SELECT BUYERID, 'is in Antiques only'
FROM ANTIQUES
WHERE BUYERID NOT IN

   (SELECT OWNERID
   FROM ORDERS);
```

The first query does a join to list any owners who are in both tables, and putting a tag line after the ID repeating the quote. The UNION merges this list with the next list. The second list is generated by first listing those ID's not in the Orders table, thus generating a list of ID's excluded from the join query. Then, each row in the Antiques table is scanned, and if the BuyerID is not in this exclusion list, it is listed with its quoted tag. There might be an easier way to make this list, but it's difficult to generate the informational quoted strings of text.

This concept is useful in situations where a primary key is related to a foreign key, but the foreign key value for some primary keys is NULL. For example, in one table, the primary key is a salesperson, and in another table is customers, with their salesperson listed in the same row. However, if a salesperson has no customers, that person's name won't appear in the customer table. The outer join is used if the listing of **all** salespersons is to be printed, listed with their customers, whether the salesperson has a customer or not--that is, no customer is printed (a logical NULL value) if the salesperson has no customers, but is in the salespersons table. Otherwise, the salesperson will be listed with each customer.

ENOUGH QUERIES!!! you say?...now on to something completely different...

Section 4.6 Miscellaneous Topics: *Embedded SQL*

an ugly example (do not write a program like this...for purposes of argument ONLY)

```
/* -To get right to it, here is an example program that uses Embedded
    SQL. Embedded SQL allows programmers to connect to a database and
    include SQL code right in the program, so that their programs can
    use, manipulate, and process data from a database.
   -This example C Program (using Embedded SQL) will print a report.
   -This program will have to be precompiled for the SQL statements,
    before regular compilation.
   -The EXEC SQL parts are the same (standard), but the surrounding C
    code will need to be changed, including the host variable
    declarations, if you are using a different language.
   -Embedded SQL changes from system to system, so, once again, check
    local documentation, especially variable declarations and logging
    in procedures, in which network, DBMS, and operating system
    considerations are crucial. */

/************************************************/
/* THIS PROGRAM IS NOT COMPILABLE OR EXECUTABLE */
/* IT IS FOR EXAMPLE PURPOSES ONLY              */
/************************************************/

#include <stdio.h>

/* This section declares the host variables; these will be the
   variables your program uses, but also the variable SQL will put
   values in or take values out. */
EXEC SQL BEGIN DECLARE SECTION;
  int BuyerID;
  char FirstName[100], LastName[100], Item[100];
EXEC SQL END DECLARE SECTION;

/* This includes the SQLCA variable, so that some error checking can be done. */
EXEC SQL INCLUDE SQLCA;

main() {

/* This is a possible way to log into the database */
EXEC SQL CONNECT UserID/Password;

/* This code either says that you are connected or checks if an error
   code was generated, meaning log in was incorrect or not possible. */
  if(sqlca.sqlcode) {
    printf(Printer, "Error connecting to database server.\n");
    exit();
  }
  printf("Connected to database server.\n");

/* This declares a "Cursor". This is used when a query returns more
   than one row, and an operation is to be performed on each row
   resulting from the query. With each row established by this query,
   I'm going to use it in the report. Later, "Fetch" will be used to
   pick off each row, one at a time, but for the query to actually
   be executed, the "Open" statement is used. The "Declare" just
   establishes the query. */
EXEC SQL DECLARE ItemCursor CURSOR FOR
  SELECT ITEM, BUYERID
  FROM ANTIQUES
  ORDER BY ITEM;
EXEC SQL OPEN ItemCursor;

/* +-- You may wish to put a similar error checking block here --+ */

/* Fetch puts the values of the "next" row of the query in the host
   variables, respectively. However, a "priming fetch" (programming
   technique) must first be done. When the cursor is out of data, a
   sqlcode will be generated allowing us to leave the loop. Notice
   that, for simplicity's sake, the loop will leave on any sqlcode,
   even if it is an error code. Otherwise, specific code checking must
   be performed. */
EXEC SQL FETCH ItemCursor INTO :Item, :BuyerID;
  while(!sqlca.sqlcode) {

/* With each row, we will also do a couple of things. First, bump the
   price up by $5 (dealer's fee) and get the buyer's name to put in
   the report. To do this, I'll use an Update and a Select, before
   printing the line on the screen. The update assumes however, that
   a given buyer has only bought one of any given item, or else the
   price will be increased too many times. Otherwise, a "RowID" logic
   would have to be used (see documentation). Also notice the colon    before host
variable names when used inside of SQL statements. */

EXEC SQL UPDATE ANTIQUES
  SET PRICE = PRICE + 5
  WHERE ITEM = :Item AND BUYERID = :BuyerID;

EXEC SQL SELECT OWNERFIRSTNAME, OWNERLASTNAME
  INTO :FirstName, :LastName
  FROM ANTIQUEOWNERS
  WHERE BUYERID = :BuyerID;

    printf("%25s %25s %25s", FirstName, LastName, Item);

/* Ugly report--for example purposes only! Get the next row. */
EXEC SQL FETCH ItemCursor INTO :Item, :BuyerID;
  }

/* Close the cursor, commit the changes (see below), and exit the
   program. */
EXEC SQL CLOSE ItemCursor;
EXEC SQL COMMIT RELEASE;
  exit();
}
```

Section 4.7 Miscellaneous Topics: *Common SQL Questions*

Advanced Topics

1. Why can't I just ask for the first three rows in a table? --Because in relational databases, rows are inserted in no particular order, that is, the system inserts them in an arbitrary order; so, you

can only request rows using valid SQL features, like ORDER BY, etc.

2. What is this DDL and DML I hear about? --DDL (Data Definition Language) refers to (in SQL) the Create Table statement...DML (Data Manipulation Language) refers to the Select, Update, Insert, and Delete statements.

3. Aren't database tables just files? --Well, DBMS's store data in files declared by system managers before new tables are created (on large systems), but the system stores the data in a special format, and may spread data from one table over several files. In the database world, a set of files created for a database is called a *tablespace*. In general, on small systems, everything about a database (definitions and all table data) is kept in one file.

4. (Related question) Aren't database tables just like spreadsheets? --No, for two reasons. First, spreadsheets can have data in a cell, but a cell is more than just a row-column-intersection. Depending on your spreadsheet software, a cell might also contain formulas and formatting, which database tables cannot have (currently). Secondly, spreadsheet cells are often dependent on the data in other cells. In databases, "cells" are independent, except that columns are logically related (hopefully; together a row of columns describe an entity), and, other than primary key and foreign key constraints, each row in a table is independent from one another.

5. How do I import a text file of data into a database? --Well, you can't do it directly...you must use a utility, such as Oracle's SQL*Loader, or write a program to load the data into the database. A program to do this would simply go through each record of a text file, break it up into columns, and do an Insert into the database.

6. What is a *schema*? --A schema is a logical set of tables, such as the Antiques database above...usually, it is thought of as simply "the database", but a database can hold more than one schema. For example, a *star schema* is a set of tables where one large, central table holds all of the important information, and is linked, via foreign keys, to *dimension* tables which hold detail information, and can be used in a join to create detailed reports.

7. What are some general tips you would give to make my SQL queries and databases better and faster (*optimized*)?

    - You should try, if you can, to avoid expressions in Selects, such as SELECT ColumnA + ColumnB, etc. The *query optimizer* of the database, the portion of the DBMS that determines the best way to get the required data out of the database itself, handles expressions in such a way that would normally require more time to retrieve the data than if columns were normally selected, and the expression itself handled programmatically.

    - Minimize the number of columns included in a Group By clause.

    - If you are using a join, try to have the columns joined on (from both tables) indexed.

    - When in doubt, index.

    - Unless doing multiple counts or a complex query, use COUNT(*) (the number of rows generated by the query) rather than COUNT(Column_Name).

8. What is *normalization*? --Normalization is a technique of database design that suggests that certain criteria be used when constructing a table layout (deciding what columns each table will

have, and creating the key structure), where the idea is to eliminate redundancy of non-key data across tables. Normalization is usually referred to in terms of *forms*, and I will introduce only the first three, even though it is somewhat common to use other, more advanced forms (fourth, fifth, Boyce-Codd; see documentation).

First Normal Form refers to moving data into separate tables where the data in each table is of a similar type, and by giving each table a primary key.

Putting data in *Second Normal Form* involves removing to other tables data that is only dependent of a part of the key. For example, if I had left the names of the Antique Owners in the items table, that would not be in Second Normal Form because that data would be redundant; the name would be repeated for each item owned; as such, the names were placed in their own table. The names themselves don't have anything to do with the items, only the identities of the buyers and sellers.

Third Normal Form involves getting rid of anything in the tables that doesn't depend solely on the primary key. Only include information that is dependent on the key, and move off data to other tables that are independent of the primary key, and create a primary keys for the new tables.

There is some redundancy to each form, and if data is in *3NF* (shorthand for 3rd normal form), it is already in *1NF* and *2NF*. In terms of data design then, arrange data so that any non-primary key columns are dependent only on the *whole primary key*. If you take a look at the sample database, you will see that the way then to navigate through the database is through joins using common key columns.

Two other important points in database design are using good, consistent, logical, full-word names for the tables and columns, and the use of full words in the database itself. On the last point, my database is lacking, as I use numeric codes for identification. It is usually best, if possible, to come up with keys that are, by themselves, self-explanatory; for example, a better key would be the first four letters of the last name and first initial of the owner, like JONEB for Bill Jones (or for tiebreaking purposes, add numbers to the end to differentiate two or more people with similar names, so you could try JONEB1, JONEB2, etc.).

9. What is the difference between a *single-row query* and a *multiple-row query* and why is it important to know the difference? --First, to cover the obvious, a single-row query is a query that returns one row as its result, and a multiple-row query is a query that returns more than one row as its result. Whether a query returns one row or more than one row is entirely dependent on the design (or *schema*) of the tables of the database. As query-writer, you must be aware of the schema, be sure to include enough conditions, and structure your SQL statement properly, so that you will get the desired result (either one row or multiple rows). For example, if you wanted to be sure that a query of the AntiqueOwners table returned only one row, consider an equal condition of the primary key-column, OwnerID.

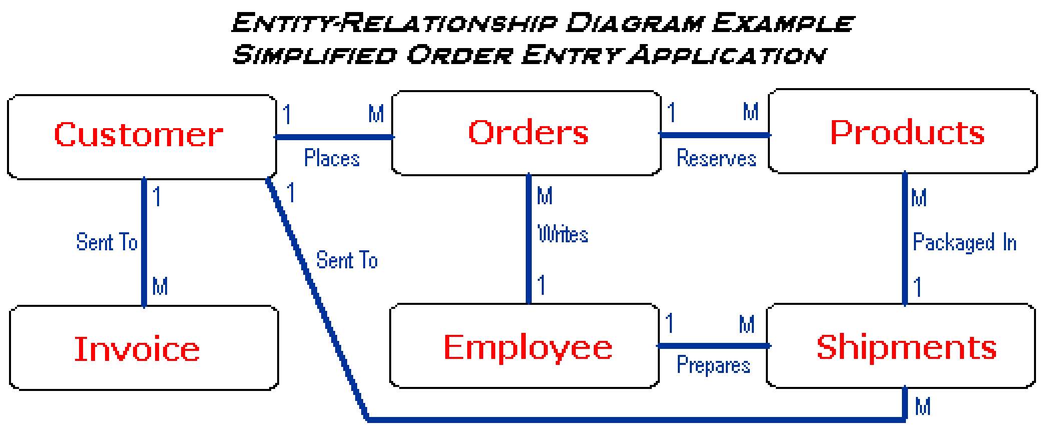
   Three reasons immediately come to mind as to why this is important. First, getting multiple rows when you were expecting only one, or vice-versa, may mean that the query is erroneous, that the database is incomplete, or simply, you learned something new about your data. Second, if you are using an update or delete statement, you had better be sure that the statement that you write performs the operation on the desired row (or rows)...or else, you might be deleting or updating more rows than you intend. Third, any queries written in Embedded SQL must be carefully thought out as to the number of rows returned. If you write a single-row query, only one SQL statement may need to be performed to complete the programming logic required. If your query, on the other hand, returns multiple rows, you will have to use the Fetch statement, and quite probably, some sort of looping structure in your program will be required to iterate processing on each returned row of the query.

10. What are *relationships?* --Another design question...the term "relationships" (often termed "relation") usually refers to the relationships among primary and foreign keys between tables. This concept is important because when the tables of a relational database are designed, these relationships must be defined because they determine which columns are or are not primary or foreign keys. You may have heard of an **Entity-Relationship Diagram**, which is a graphical view of tables in a database schema, with lines connecting related columns across tables. See the sample diagram at the end of this section or some of the sites below in regard to this topic, as there are many different ways of drawing E-R diagrams. But first, let's look at each kind of relationship...

    A *One-to-one relationship* means that you have a primary key column that is related to a foreign key column, and that for every primary key value, there is **one** foreign key value. For example, in the first example, the EmployeeAddressTable, we add an EmployeeIDNo column. Then, the EmployeeAddressTable is related to the EmployeeStatisticsTable (second example table) by means of that EmployeeIDNo. Specifically, each employee in the EmployeeAddressTable **has** statistics (one row of data) in the EmployeeStatisticsTable. Even though this is a contrived example, this is a "1-1" relationship. Also notice the "has" in bold...when expressing a relationship, it is important to describe the relationship with a verb.

    The other two kinds of relationships may or may not use logical primary key and foreign key constraints...it is strictly a call of the designer. The first of these is the *one-to-many relationship* ("1-M"). This means that for every column value in one table, there is **one or more** related values in another table. Key constraints may be added to the design, or possibly just the use of some sort of identifier column may be used to establish the relationship. An example would be that for every OwnerID in the AntiqueOwners table, there are one or more (zero is permissible too) Items **bought** in the Antiques table (verb: buy).

    Finally, the *many-to-many relationship* ("M-M") does not involve keys generally, and usually involves idenifying columns. The unusual occurence of a "M-M" means that one column in one table is related to another column in another table, and for every value of one of these two columns, there are one or more related values in the corresponding column in the other table (and vice-versa), or more a common possibility, two tables have a 1-M relationship to each other (two relationships, one 1-M going each way). A [bad] example of the more common situation would be if you had a job assignment database, where one table held one row for each employee and a job assignment, and another table held one row for each job with one of the assigned employees. Here, you would have multiple rows for each employee in the first table, one for each job assignment, and multiple rows for each job in the second table, one for each employee assigned to the project. These tables have a M-M: each employee in the first table **has** many job assignments from the second table, and each job **has** many employees assigned to it from the first table. This is the tip of the iceberg on this topic...see the links below for more information and see the diagram below for a *simplified* example of an E-R diagram.



Read a relationship as: A Shipment is "sent to" a customer.
For one Customer, there can be one or more ("many") shipments.

11. What are some important nonstandard SQL features (extremely common question)? --Well, see the next section...

Section 4.8 Miscellaneous Topics: *Nonstandard SQL*

"check local listings"

- INTERSECT and MINUS are like the UNION statement, except that INTERSECT produces rows that appear in both queries, and MINUS produces rows that result from the first query, but not the second.
- Report Generation Features: the COMPUTE clause is placed at the end of a query to place the result of an aggregate function at the end of a listing, like **`COMPUTE SUM (PRICE);`** Another option is to use break logic: define a break to divide the query results into groups based on a column, like **`BREAK ON BUYERID`**. Then, to produce a result after the listing of a group, use **`COMPUTE SUM OF PRICE ON BUYERID`**. If, for example, you used all three of these clauses (BREAK first, COMPUTE on break second, COMPUTE overall sum third), you would get a report that grouped items by their BuyerID, listing the sum of Prices after each group of a BuyerID's items, then, after all groups are listed, the sum of all Prices is listed, all with SQL-generated headers and lines.
- In addition to the above listed aggregate functions, some DBMS's allow more functions to be used in Select lists, except that these functions (some character functions allow multiple-row results) are to be used with an individual value (not groups), on *single-row queries*. The functions are to be used only on appropriate data types, also. Here are some **Mathematical Functions**:

ABS(X)	Absolute value-converts negative numbers to positive, or leaves positive numbers

	alone
CEIL(X)	X is a decimal value that will be rounded up.
FLOOR(X)	X is a decimal value that will be rounded down.
GREATEST(X,Y)	Returns the largest of the two values.
LEAST(X, Y)	Returns the smallest of the two values.
MOD(X,Y)	Returns the remainder of X / Y.
POWER(X, Y)	Returns X to the power of Y.
ROUND(X, Y)	Rounds X to Y decimal places. If Y is omitted, X is rounded to the nearest integer.
SIGN(X)	Returns a minus if X < 0, else a plus.
SQRT(X)	Returns the square root of X.

Character Functions

LEFT(<string>,X)	Returns the leftmost X characters of the string.
RIGHT(<string>,X)	Returns the rightmost X characters of the string.
UPPER(<string>)	Converts the string to all uppercase letters.
LOWER(<string>)	Converts the string to all lowercase letters.
INITCAP(<string>)	Converts the string to initial caps.
LENGTH(<string>)	Returns the number of characters in the string.
<string>||<string>	Combines the two strings of text into one, *concatenated* string, where the first string is immediately followed by the second.

LPAD(<string>,X,'*')	Pads the string on the left with the * (or whatever character is inside the quotes), to make the string X characters long.
RPAD(<string>,X,'*')	Pads the string on the right with the * (or whatever character is inside the quotes), to make the string X characters long.
SUBSTR(<string>,X,Y)	Extracts Y letters from the string beginning at position X.
NVL(<column>,<value>)	The Null value function will substitute <value> for any NULLs for in the <column>. If the current value of <column> is not NULL, NVL has no effect.

Section 4.9 Miscellaneous Topics: *Syntax Summary*

~ For Advanced Users Only

Here are the general forms of the statements discussed in this tutorial, plus some extra important ones (explanations given). **REMEMBER** that all of these statements may or may not be available on your system, so check documentation regarding availability:

`ALTER TABLE <TABLE NAME> ADD|DROP|MODIFY (COLUMN SPECIFICATION[S]...see Create Table);` --allows you to add or delete a column or columns from a table, or change the specification (data type, etc.) on an existing column; this statement is also used to change the physical specifications of a table (how a table is stored, etc.), but these definitions are DBMS-specific, so read the documentation. Also, these physical specifications are used with the Create Table statement, when a table is first created. In addition, only one option can be performed per Alter Table statement--either add, drop, **OR** modify in a single statement.

`COMMIT;` --makes changes made to some database systems permanent (since the last COMMIT; known as a *transaction*)

`CREATE [UNIQUE] INDEX <INDEX NAME>`
`ON <TABLE NAME> (<COLUMN LIST>);` --UNIQUE is optional; within brackets.

`CREATE TABLE <TABLE NAME>`
`(<COLUMN NAME> <DATA TYPE> [(<SIZE>)] <COLUMN CONSTRAINT>,`
`...other columns);` (also valid with ALTER TABLE)
--where SIZE is only used on certain data types (see above), and constraints include the following possibilities (automatically enforced by the DBMS; failure causes an error to be generated):

1. NULL or NOT NULL (see above)
2. UNIQUE enforces that no two rows will have the same value for this column

3. PRIMARY KEY tells the database that this column is the primary key column (only used if the key is a one column key, otherwise a PRIMARY KEY (column, column, ...) statement appears after the last column definition.

4. CHECK allows a condition to be checked for when data in that column is updated or inserted; for example, **`CHECK (PRICE > 0)`** causes the system to check that the Price column is greater than zero before accepting the value...sometimes implemented as the CONSTRAINT statement.

5. DEFAULT inserts the default value into the database if a row is inserted without that column's data being inserted; for example, **`BENEFITS INTEGER DEFAULT = 10000`**

6. FOREIGN KEY works the same as Primary Key, but is followed by: **`REFERENCES <TABLE NAME> (<COLUMN NAME>)`**, which refers to the referential primary key.

```
CREATE VIEW <TABLE NAME> AS <QUERY>;
```

```
DELETE FROM <TABLE NAME> WHERE <CONDITION>;
```

```
INSERT INTO <TABLE NAME> [(<COLUMN LIST>)]
VALUES (<VALUE LIST>);
```

`ROLLBACK;` --Takes back any changes to the database that you have made, back to the last time you gave a Commit command...beware! Some software uses automatic committing on systems that use the transaction features, so the Rollback command may not work.

```
SELECT [DISTINCT|ALL] <LIST OF COLUMNS, FUNCTIONS, CONSTANTS, ETC.>
FROM <LIST OF TABLES OR VIEWS>
[WHERE <CONDITION(S)>]
[GROUP BY <GROUPING COLUMN(S)>]
[HAVING <CONDITION>]
```
`[ORDER BY <ORDERING COLUMN(S)> [ASC|DESC]];` --where ASC|DESC allows the ordering to be done in ASCending or DESCending order

```
UPDATE <TABLE NAME>
SET <COLUMN NAME> = <VALUE>
```
`[WHERE <CONDITION>];` --if the Where clause is left out, all rows will be updated according to the Set statement

Disclaimer

I hope you have learned something from this introductory look at a very important language that is becoming more prevalent in the world of client-server computing. I wrote this tutorial in order to contribute something of value to the web and the web community. In fact, I have been informed that this document is being used at several colleges for use in database classes and for use by researchers. Also, look for this page in Waite Publishing's book about Borland C++ Builder. In addition, I would like to thank all of the people from across five continents who have contacted me regarding this web page.

I also hope to continue to add more material to this tutorial, such as topics about database design and nonstandard SQL extensions, even though I wish to stay away from material about individual Database Management Systems. Good luck in your SQL and other computing adventures.

~ Jim Hoffman