SQL Tutorial

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.

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

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 (').

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);
```

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.

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 that relative position 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.

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

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.

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.

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.*

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.

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).

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.

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.

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.

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');
```

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;
```

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.

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;
```

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 OWNERID
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 =

    (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*).

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.).

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.

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 ANTIQUEOWNERS
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...

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