SQL - Tables

Data is stored inside **SQL tables** which are contained within **SQL databases**. A single database can house hundreds of tables, each playing its own unique role in th+e database schema. While database architecture and schema are concepts far above the scope of this tutorial, we plan on diving in just past the surface to give you a glimpse of database architecture that begins with a thorough understanding of SQL Tables.

SQL tables are comprised of table **rows** and **columns**. Table columns are responsible for storing many different types of data, like numbers, texts, dates, and even files. There are many different types of table columns and these **data types** vary, depending on how the SQL table has been created by the SQL developer. A table row is a horizontal record of values that fit into each different table column.

SQL - Create a SQL Table

Let's now ***CREATE*** a SQL table to help us expand our knowledge of SQL and SQL commands. This new table will serve as a practice table and we will begin to populate this table with some data which we can then manipulate as more SQL Query commands are introduced. The next couple of examples will definitely be overwhelming to novice SQL programmers, but we will take a moment to explain what's going on.

SQL Create Table Query:

```
USE mydatabase;

CREATE TABLE orders
(id INT IDENTITY(1,1) PRIMARY KEY,
customer VARCHAR(50),
day_of_order DATETIME,
product VARCHAR(50),
quantity INT);
```

The first line of the example, "USE mydatabase;", is pretty straightforward. This line defines the **query scope** and directs SQL to run the command against the *MyDatabase* object we created earlier in the SQL Databases lesson. The blank line break after the first command is not required, but it makes our query easier to follow. The line starting with the ***CREATE*** clause is where we are actually going to tell SQL to create the new table, which is named *orders*.

Each table column has its own set of guidelines or schema, and the lines of code above contained in parenthesis () are telling SQL how to go about setting up each column schema. Table columns are presented in list format, and each schema is separated with a comma (,). It isn't important to fully understand exactly what all of these schema details mean just yet. They will be explained in more detail throughout the remainder of the tutorial. For now, just take note that we are creating a new, empty SQL table named ***orders***, and this table is 5 columns wide.

SQL - INSERT DATA into your New Table

Next, we will use SQL's ***INSERT*** command to draw up a query that will insert a new data row into our brand new SQL table, ***orders***. If you're already familiar with everything we've covered so far, please execute the query below and then skip ahead and start learning about other SQL Queries.

SQL Insert Query:

```
USE mydatabase;

INSERT INTO orders
(customer,day_of_order,product, quantity)
VALUES('Tizag','8/1/08','Pen',4);
```

SQL Insert Query Results:

(1 row(s) affected)

This message ("1 row(s) affected") indicates that our query has run successfully and also informs us that 1 row has been affected by the query. This is the desired result as our goal was to insert a single record into the newly formed ***orders*** table.

Listed above is a typical ***INSERT*** query used to insert data into the table we had previously created. The first line ("USE mydatabase;") identifies the query scope and the line after indicates what it is we'd like SQL to do for us. ("INSERT INTO orders") inserts data into the ***orders*** table. Then, we have to list each table column by name (customer,day_of_order,product, quantity) and finally provide a list of values to insert into each table column VALUES('Tizag','8/1/08','Pen',4).

You may notice that we have not included the ***id*** column, and this is intentional. We have set this column up in a way that allows SQL to populate this field automatically, and therefore, we do not need to worry about including it in any of our ***INSERT*** statements.

SQL - Create a SQL Table

Let's now ***CREATE*** a SQL table to help us expand our knowledge of SQL and SQL commands. This new table will serve as a practice table and we will begin to populate this table with some data which we can then manipulate as more SQL Query commands are introduced. The next couple of examples will definitely be overwhelming to novice SQL programmers, but we will take a moment to explain what's going on.

SQL Create Table Query:

```
USE mydatabase;

CREATE TABLE orders
(id INT IDENTITY(1,1) PRIMARY KEY,
customer VARCHAR(50),
day_of_order DATETIME,
product VARCHAR(50),
quantity INT);
```

The first line of the example, "USE mydatabase;", is pretty straightforward. This line defines the **query scope** and directs SQL to run the command against the *MyDatabase* object we created earlier in the SQL Databases lesson. The blank line break after the first command is not required, but it makes our query easier to follow. The line starting with the ***CREATE*** clause is where we are actually going to tell SQL to create the new table, which is named *orders*.

Each table column has its own set of guidelines or schema, and the lines of code above contained in parenthesis () are telling SQL how to go about setting up each column schema. Table columns are presented in list format, and each schema is separated with a comma (,). It isn't important to fully understand exactly what all of these schema details mean just yet. They will be explained in more detail throughout the remainder of the tutorial. For now, just take note that we are creating a new, empty SQL table named ***orders***, and this table is 5 columns wide.

SQL - INSERT DATA into your New Table

Next, we will use SQL's ***INSERT*** command to draw up a query that will insert a new data row into our brand new SQL table, ***orders***. If you're already familiar with everything we've covered so far, please execute the query below and then skip ahead and start learning about other SQL Queries.

SQL Insert Query:

```
USE mydatabase;

INSERT INTO orders
(customer,day_of_order,product, quantity)
VALUES('Tizag','8/1/08','Pen',4);
```

SQL Insert Query Results:

(1 row(s) affected)

This message ("1 row(s) affected") indicates that our query has run successfully and also informs us that 1 row has been affected by the query. This is the desired result as our goal was to insert a single record into the newly formed ***orders*** table.

Listed above is a typical ***INSERT*** query used to insert data into the table we had previously created. The first line ("USE mydatabase;") identifies the query scope and the line after indicates what it is we'd like SQL to do for us. ("INSERT INTO orders") inserts data into the ***orders*** table. Then, we have to list each table column by name (customer,day_of_order,product, quantity) and finally provide a list of values to insert into each table column VALUES('Tizag','8/1/08','Pen',4).

You may notice that we have not included the ***id*** column, and this is intentional. We have set this column up in a way that allows SQL to populate this field automatically, and therefore, we do not need to worry about including it in any of our ***INSERT*** statements.

ELECT queries are the most commonly used SQL commands, so let's take a look at a ***SELECT*** query that will return records from the ***orders*** table that we created previously in the SQL Tables lesson.

SQL Query Code:

```
USE mydatabase;

SELECT * FROM orders;
```

SQL Query Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4

We'll explain the mechanics of this code in the next lesson. For now, just know that ***SELECT*** queries essentially tell SQL to go and "fetch" table data for your viewing pleasure.

Here's a look at a few different query types including a ***INSERT*** and ***SELECT*** query we will be covering in the next lesson, SQL Select.

SQL Query Examples:

```
-- Inserts data into a SQL Database/Table
INSERT INTO orders (customer,day_of_order,product, quantity)
VALUES('Tizag','8/1/08','Pen',4);

-- Selects data from a SQL Database/Table
SELECT * FROM orders;

-- Updates data in a Database/Table
UPDATE orders SET quantity = '6'
WHERE id = '1'
```

SQL - Alter

SQL ***ALTER*** is the command used to add, edit, and modify data objects like tables, databases, and views. ***ALTER*** is the command responsible for making table column adjustments or renaming table columns. New table columns can also be added and dropped from existing SQL tables.

SQL Add:

```
USE mydatabase;

ALTER TABLE orders
ADD discount VARCHAR(10);
```

SQL Results:

id	customer	day_of_order	product	quantity	discount
1	Tizag	2008-08-01 00:00:00.000	Pen	8	NULL
2	Tizag	2008-08-01 00:00:00.000	Stapler	3	NULL
3	A+Maintenance	2008-08-16 00:00:00.000	Hanging Files	14	NULL
4	Gerald Garner	2008-08-15 00:00:00.000	19" LCD Screen	5	NULL
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	5	NULL
6	Tizag	2008-07-25 00:00:00.000	HP Printer	4	NULL

As you can see from the results panel, SQL has added an additional column, discount, to the orders table. Since this column was just created, it contains no data, and only *NULL* values have been returned.

SQL - Alter Table: Modify Column

SQL table columns can be altered and changed using the ***MODIFY COLUMN*** command. This allows the developer the opportunity to mold table columns or adjust settings as needed.

SQL Modify Column:

```
USE mydatabase;

ALTER TABLE orders
ALTER COLUMN discount DECIMAL(18,2);
```

Above, we have modified the new *discount* table column changing the column data type from a *varchar* to a *decimal* table column. This example can be expanded to modify table columns as needed by the developer.

SQL - SQL Alter Table: Drop

This column can be deleted using the SQL ***DROP*** command. Once this column has been dropped, however, the data stored inside of it will be lost forever. Proceed with caution!

SQL Drop Column Code:

```
USE mydatabase;

ALTER TABLE orders
DROP COLUMN discount;
```

SQL - Between

BETWEEN is a conditional statement found in the ***WHERE*** clause. It is used to query for table rows that meet a condition falling between a specified range of numeric values. It would be used to answer questions like, "How many orders did we receive ***BETWEEN*** July 20th and August 5th?"

SQL Select Between:

```
USE mydatabase;

SELECT *
FROM orders
WHERE day_of_order BETWEEN '7/20/08' AND '8/05/08';
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4
2	Tizag	2008-08-01 00:00:00.000	Stapler	1
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3
6	Tizag	2008-07-25 00:00:00.000	HP Printer	2

BETWEEN essentially combines two conditional statements into one and simplifies the querying process for you. To understand exactly what we mean, we could create another query without using the ***BETWEEN*** condition and still come up with the same results, (using AND instead).

SQL Select Between:

```
USE mydatabase;

SELECT *
FROM orders
WHERE day_of_order >= '7/20/08'
AND day_of_order <= '8/05/08';
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4
2	Tizag	2008-08-01 00:00:00.000	Stapler	1
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3
6	Tizag	2008-07-25 00:00:00.000	HP Printer	2

As you can see from comparing the results of these two queries, we are able to retrieve the same data, but you may find ***BETWEEN*** easier to use and less cumbersome than writing two different conditional statements. In the end, the preference is really up to the individual writing the SQL Code

SQL - Operators

SQL **operators** are found in just about every SQL query. Operators are the mathematical and equality symbols used to compare, evaluate, or calculate values. Equality operators include the (<), (>), and (=) symbols, which are used to compare one value against another. Each of these characters have special meaning, and when SQL comes across them, they help tell SQL how to evaluate an expression or conditional statement. Most operators will appear inside of conditional statements in the ***WHERE*** clause of SQL Commands.

Operators come in three flavors: mathematical, logical, and equality. Mathematical operators add, subtract, multiply, and divide numbers. Logical operators include ***AND*** and ***OR***. Take note of the following tables for future reference.

SQL operators are generally found inside of queries-- more specifically, in the conditional statements of the ***WHERE*** clause.

SQL Equality Operator Query:

```
USE mydatabase;

SELECT customer,day_of_order
FROM orders
WHERE  day_of_order > '7/31/08'
```

Sql Equality Operator:

customer	day_of_order
Tizag	2008-08-01 00:00:00.000
Tizag	2008-08-01 00:00:00.000

In this case, we've used the equality operator ***greater than*** (>) to return orders from the orders table with a date greater than ***'7/31/08'***.

SQL - Equality Operator Table

Equality involves comparing two values. To do so requires the use of the (<), (>), or (=) special characters. Does X = Y? Is Y < X? These are both questions that can be answered using a SQL Equality Operator expression.

SQL Equality Operators:

Operator	Example	Defined	Result
=, IS	5 = 5	5 equal to 5?	True
!=, IS NOT	7 != 2	7 IS NOT (!=) equal to 2?	True
<	7 < 4	7 less than 4?	False
>	7 > 4	greater than 4?	True

<=	7 <= 11	Is 7 less than or equal to 11?	True
>=	7 >= 11	Is 7 greater than or equal to 11?	False

SQL - Mathematical Operators

SQL mathematical operations are performed using mathematical operators (+, -, *, /, and %). We can use SQL like a calculator to get a feel for how these operators work.

SQL Mathematical Operators:

```
SELECT
15 + 4, --Addition
15 - 4, --Subtraction
15 * 4, --Multiplication
15 / 5, -- Division
15 % 4; --Modulus
```

SQL Results:

Addition	Subtraction	Multiplication	Division	Modulus
19	11	60	3	3

Modulus may be the only unfamiliar term on the chart. Modulus performs division, dividing the first digit by the second digit, but instead of returning a quotient, a "*remainder*" value is returned instead.

Modulus Example:

```
USE mydatabase;

SELECT (5 / 2) -- = 2.5
SELECT (5 % 2) -- = 1 is the value that will be returned
```

SQL - Logical Operators

These operators provide you with a way to specify exactly what you want SQL to go and fetch, and you may be as specific as you'd like! We'll discuss these a little later on and provide some real world scenarios as well.

We cover these operators thoroughly in the SQL AND/OR lesson.

- AND - Compares/Associates two values or expressions

- OR - Compares/Associates two values or expressions

SQL - And

SQL ***AND*** links together two or more conditional statements for increased filtering when running SQL commands. ***AND*** helps the developer query for very specific records while answering questions like, "I want to view all orders made by a certain customer AND made on a special date." There is no limit to the number of ***AND/OR*** conditions that can be applied to a query utilizing the ***WHERE*** clause. This makes it possible for the developer to be as precise as needed when querying for results.

SQL And Code:

```
USE mydatabase;

SELECT *
FROM orders
WHERE customer = 'Tizag'
AND day_of_order = '08/01/08'
AND product = 'Pen';
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4

This example illustrates how SQL ***AND*** combines multiple conditional statements (3 total now) into a single condition with multiple circumstances (filters). Each filter removes rows from the result set that doesn't meet the condition.

SQL - Or

SQL ***OR*** also applies logic to help filter results. The difference is that instead of linking together conditional statements, an OR condition just asks SQL to look for 2 separate conditions within the same query and return any records/rows matching either of the conditions.

SQL Or Code:

```
USE mydatabase;

SELECT *
FROM orders
WHERE product = 'Pen'
```

```
OR product = '19" LCD Screen';
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4
4	Gerald Garner	2008-08-15 00:00:00.000	19" LCD Screen	3
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3

The first record returned matches the first condition since the product = 'Pen'. The two records after that match the other condition; the product in each of those orders is the '19" LCD Screen'. This type of logic allows the developer to filter results based on one or more conditions.

SQL ***AND*** allows the developer to query for more specific data by linking together conditional statements. On the other end of the spectrum, SQL ***OR*** creates a new, independent condition not linked to any other conditional statement.

SQL - And / Or Combination

Conditional statements can be grouped together using parentheses (). Doing so links together conditions and provides robust solutions for data querying.

SQL And/Or Code:

```
USE mydatabase;

SELECT *
FROM orders
WHERE (quantity > 2 AND customer = 'Tizag')
OR (quantity > 0 AND customer = 'Gerald Garner')
```

By encapsulating the first two conditions (quantity > 2 AND customer = 'Tizag') SQL now treats this as a single condition, and the same is true for the next line. These two conditions have been linked together with the ***OR*** operator, creating very unique behavior.

SQL is now looking for rows where the *customer* is *Tizag* AND the *quantity* is more than *2*, and ALSO looking for rows where the *customer* is *Gerald Garner* AND the *quantity* is greater than *0*. All rows meeting either condition will be returned as demonstrated in our results below.

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4
4	Gerald Garner	2008-08-15 00:00:00.000	19" LCD Screen	3
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3

SQL - Distinct

SQL ***SELECT DISTINCT*** is a very useful way to eliminate retrieving duplicate data reserved for very specific situations. To understand when to use the ***DISTINCT*** command, let's look at a real world example where this tool will certainly come in handy.

If you've been following along in the tutorial, we have created an orders table with some data inside that represents different orders made by some of our very loyal customers over a given time period. Let's pretend that we have just heard word from our preferred shipping agent that orders made in August require no shipping charges, and we now have to notify our customers. We do not want to send mailers to all of our customers, just the ones that have placed orders in August. Also, we want to avoid retrieving duplicate customers as our customers may have placed more than one order during the month of August.

We can write a very simple SQL query to extract this information from the orders table:

SQL Select Distinct:

```
USE mydatabase;

SELECT DISTINCT customer
FROM orders
WHERE day_of_order BETWEEN '7/31/08' AND '9/1/08';
```

SQL Results:

customer
A+Maintenance
Gerald Garner

Tizag

Running this query yields a list of all the customer's affected by our unexpected news from the shipping agency. With this list, we can now go about contacting each of these customers and informing them of the good news without worrying about contacting the same customer multiple times.

SQL - Update

SQL ***UPDATE*** is the command used to update existing table rows with new data values. ***UPDATE*** is a very powerful command in the SQL world. It has the ability to update every single row in a database with the execution of only a single query. Due to ***UPDATE's*** supreme authority, it is in your best interest to always include a ***WHERE*** clause when working with ***UPDATE*** query statements. That way, you will not accidentally update more rows than you intend to.

Execute the following ***UPDATE*** command to update the customer *orders* table. Since we've provided a ***WHERE*** condition with this update command, this update will only modify rows that match the condition and in this case it happens to be *order* number *1* made by *Tizag*. This update should increase the quantity from *4 Pens* to *6 Pens* for *Tizag's* first order.

SQL Update Query:

```
USE mydatabase;

UPDATE orders
SET quantity = '6'
WHERE id = '1'
```

SQL Results:

(1 row(s) affected)

Let's verify our results by selecting this row from the *orders* table.

SQL Verification Query:

```
USE mydatabase;

SELECT *
FROM orders
WHERE id = '1'
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	6

The *orders* table now indicates that the customer *Tizag* will be ordering *6 Pens* instead of *4*. If the ***WHERE*** condition is removed from this statement, SQL would modify every row with the new *quantity* value of *6* instead of just the single row that meets the condition of *id = "1"*. SQL ***UPDATE*** replaces data, much like overwriting a previously saved file on a computer hard drive. Once you click "Save," the old file is lost and replaced with the new file. Once an ***UPDATE*** command has been executed, the old data values are lost, being overwritten by the new value.

SQL - Update Incrementing a Value

In the previous example, an order quantity was updated from *4* to *6*. Say what we really wanted to do was not necessarily change it to *6*, but to add 2 to the original order quantity. Updating the order quantity from 4 to 6 might have gotten the job done in that scenario, but that solution doesn't scale well. In the long run, we wouldn't get very much "bang for our buck," as they say.

So, perhaps a better way to tackle the same problem would be to increment the existing value (add 2) rather than updating with a single, static value. So, instead of setting the *quantity* table column to a specific value of *6*, we can send it the current table column value directly and then add 2 to that already existing value.

SQL Update Code:

```
USE mydatabase;

UPDATE orders
SET quantity = (quantity + 2)
WHERE id = '1'
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	8

Executing this update statement instead of the first update query is a huge timesaver. We no longer need to know the quantity of the order beforehand and we can add or subtract values from it in its current state. All we need

SQL - Order By

ORDER BY is the SQL command used to sort rows as they are returned from a ***SELECT*** query. SQL order by command may be added to the end of any select query and it requires at least one table column to be specified in order for SQL to sort the results.

SQL Order by query:

```
USE mydatabase;

SELECT *
FROM orders
WHERE customer = 'Tizag'
ORDER BY day_of_order;
```

Executing this query should offer a list of orders made by Tizag and you may noticed that the result set has now been sorted (low to high) according to the date value. In other words, the oldest order to the newest order.

SQL Results:

id	customer	day_of_order	product	quantity
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3
6	Tizag	2008-07-25 00:00:00.000	HP Printer	2
1	Tizag	2008-08-01 00:00:00.000	Pen	4
2	Tizag	2008-08-01 00:00:00.000	Stapler	1

SQL - Ascending & Descending

The default sort order for ***ORDER BY*** is an ascending list, [a - z] for characters or [0 - 9] for numbers. As an alternative to the default sorting for our results, which is ***ASCENDING*** (ASC), we can instead tell SQL to order the table columns in a ***DESCENDING*** (DESC) fashion [z-a].

SQL Order by Descending:

```
USE mydatabase;

SELECT *
FROM orders
WHERE customer = 'Tizag'
ORDER BY day_of_order DESC
```

SQL Results:

id	customer	day_of_order	product	quantity
1	Tizag	2008-08-01 00:00:00.000	Pen	4
2	Tizag	2008-08-01 00:00:00.000	Stapler	1
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3
6	Tizag	2008-07-25 00:00:00.000	HP Printer	2

If you compare these results to the results above, you should notice that we've pulled the same information but it is now arranged in a reverse (descending) order.

SQL - Sorting on Multiple Columns

Results may be sorted on more than one column by listing multiple column names in the ***ORDER BY*** clause, similar to how we would list column names in each ***SELECT*** statement.

SQL Order by Multiple columns:

```
USE mydatabase;

SELECT *
FROM orders
ORDER BY customer, day_of_order;
```

This query should alphabetize by customer, grouping together orders made by the same customer and then by the purchase date. SQL sorts according to how the column names are listed in the ***ORDER BY*** clause.

SQL Results:

id	customer	day_of_order	product	quantity
3	A+Maintenance	2008-08-16 00:00:00.000	Hanging Files	12
4	Gerald Garner	2008-08-15 00:00:00.000	19" LCD Screen	3
5	Tizag	2008-07-25 00:00:00.000	19" LCD Screen	3
6	Tizag	2008-07-25 00:00:00.000	HP Printer	2

1	Tizag	2008-08-01 00:00:00.000	Pen	4
2	Tizag	2008-08-01 00:00:00.000	Stapler	1

SQL - Subqueries

Subqueries are query statements tucked inside of query statements. Like the order of operations from your high school Algebra class, order of operations also come into play when you start to embed SQL commands inside of other SQL commands (subqueries). Let's take a look at a real world example involving the *orders* table and figure out how to select only the most recent order(s) in our orders table.

To accomplish this, we are first going to introduce a built-in SQL function, MAX(). This function wraps around a table column and quickly returns the current highest (max) value for the specified column. We are going to use this function to return the current "highest", aka most recent date value in the *orders* table.

SQL Subquery Preview:

```
USE mydatabase;

SELECT MAX(day_of_order)
FROM orders
```

SQL Results:

day_of_order
2008-08-16 00:00:00.000

Now we can throw this query into the ***WHERE*** clause of another ***SELECT*** query and obtain the results to our little dilemma.

SQL Select Subquery Code:

```
USE mydatabase;

SELECT *
FROM orders
WHERE day_of_order = (SELECT MAX(day_of_order) FROM orders)
```

:

id	customer	day_of_order	product	quantity

3	A+Maintenance	2008-08-16 00:00:00.000	Hanging Files	14

This query is a **dynamic query** as it pulls current information and will change if a new order is placed. Utilizing a subquery we were able to build a dynamic and robust solution for providing us with current order information.