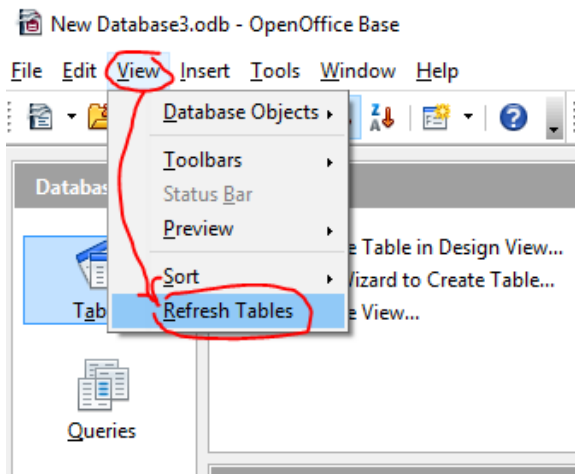


Basic SQL Tutorial, using Apache's OpenOffice Base

This tutorial will explain Structured Query Language (**SQL**) and basic database design using Apache OpenOffice Base, an open source Database Management System (**DBMS**).

This tutorial serves only as a basic overview of SQL and therefore is not intended to be used as a comprehensive SQL guide. While reading this document, if there are any terms that remain unfamiliar to you, please take a few moments to research more about those particular terms.

***NOTE** – If you are copying and pasting from this tutorial, be sure that the character encoding is correct as often times characters are NOT the same from word documents as the programming language you may be working with. For instance, Microsoft Word uses ' and " but OpenOffice Base requires ' and " or there will be an errors or omitted data.



***NOTE** – OpenOffice Base has a slight bug, sometimes you will need to hit the refresh button before some of our queries will be fully recognized.

What this drill already assumes

This drill assumes that you already have a current version of OpenOffice Base installed. At the time this document was being composed, the current version for OpenOffice Base was 4.0.1.

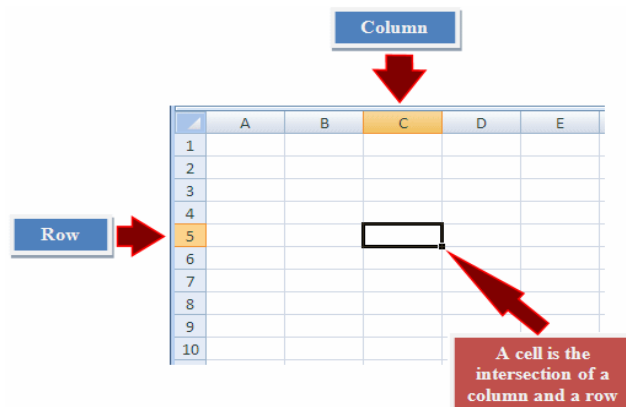
Basic Database Vocabulary

Row - refers to a Horizontal row of data in a table making up a **record** or **record set**.

record set	Name	Email Address	Phone Number	Relationship
→	Bill	bill@yahoo.com	(102) 399-2893	Friend

Column - the vertical orientation of data in a data base commonly referred to as a field.

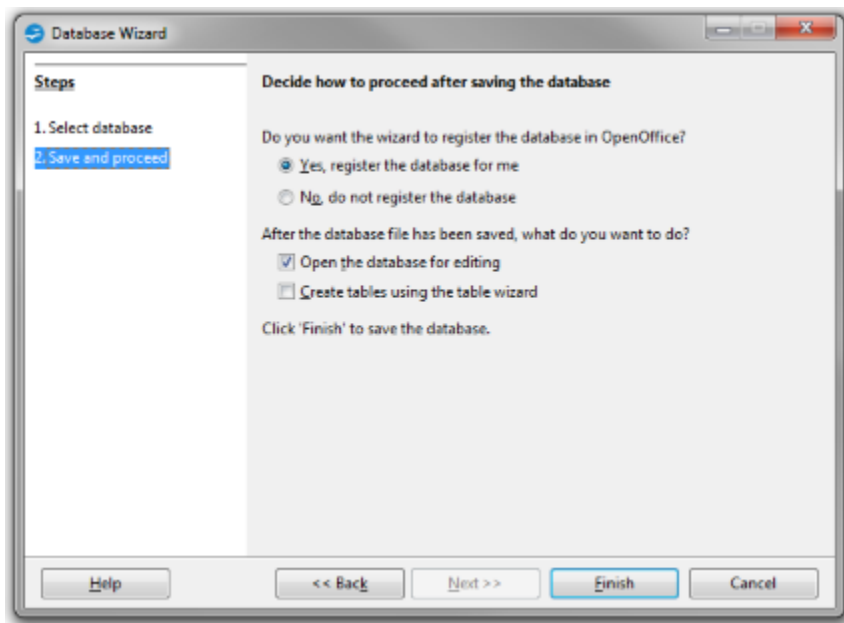
Cell - where the record set's row intersects with the corresponding column.



Creating our database

Now that you have a better understanding of the anatomy of the SQL statement, let's apply this as we will need to create a database.

After the Apache OpenOffice Base (OOB) is installed it will bring you to the OpenOffice Database Wizard. Select **"Create a new database"**, select **"Yes, register the database for me"**, select **"Open the database for me"**, click **"Finish"**.

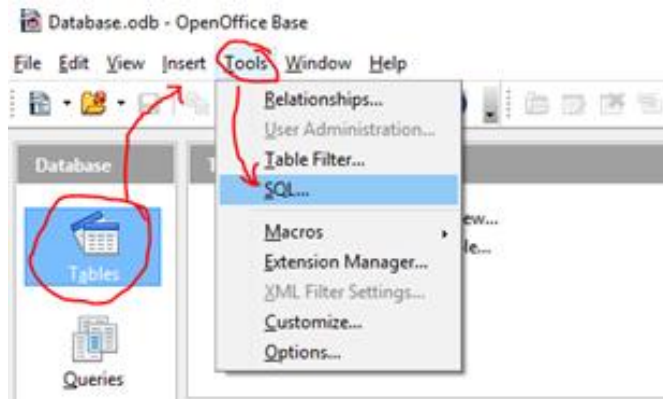


Base will ask you to save the new database. Name it, **"Database"** and then save it on your desktop or in your working directory for this tutorial.

Creating tables for our database

The first SQL statement to learn is the **CREATE TABLE** statement. The **CREATE TABLE** statement will instruct the DBMS that it will need to create a new table in the database. This statement will also contain additional parameters for exactly how to build the table.

From the main window within OpenOffice Base, click on the “**Tables**” icon in the left menu pane and then in the top menu ribbon, select “**Tools**”, and finally select “**SQL...**”

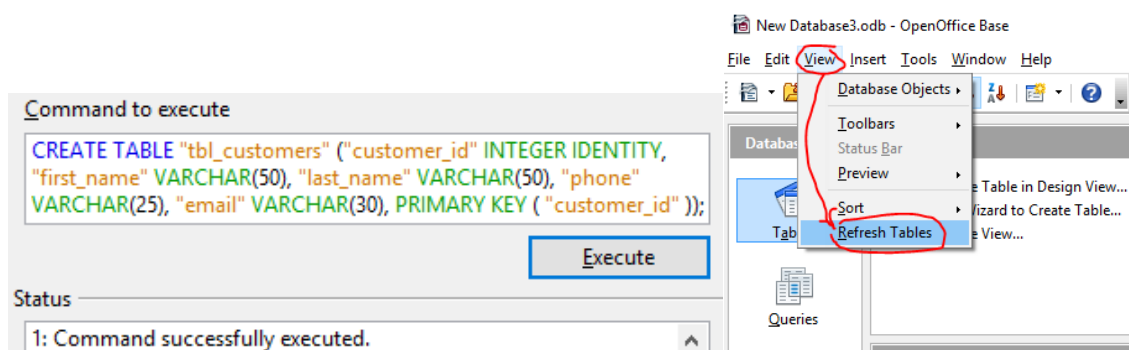


An SQL command window should appear. We can use this window to type out most of our SQL command statements. **SELECT** queries will be entered in a completely different location and will be discussed later in this tutorial.

From the SQL command window, type in the following **CREATE TABLE** statement and press “**Execute**”.

```
CREATE TABLE "tbl_customers" ("customer_id" INTEGER IDENTITY, "first_name" VARCHAR(50),  
"last_name" VARCHAR(50), "phone" VARCHAR(25), "email" VARCHAR(30), PRIMARY KEY ( "customer_id" ) );
```

***NOTE** - Sometimes you will need to click on “**Refresh Tables**”. If you receive an error, check your grammar and spelling. SQL Syntax for Base is very particular and it is good practice to learn its nuances.



The Create Table statement breakdown

CREATE TABLE “tbl_customers”

- This instructs the DBMS to create a table with the name “tbl_customers”.

```
("customer_id" INTEGER IDENTITY, "first_name" VARCHAR(50), "last_name" VARCHAR(50), "phone"
INTEGER, "email" VARCHAR(30),
```

- This part of the statement instructs the DBMS exactly what the fields will be as denoted with orange syntax. The green syntax specifies the data type that the field will require and how much memory needs to be allocated for that field of data. The IDENTITY syntax is defaulted to (1,1) which is an Auto INCREMENTING field in OpenOffice.

```
PRIMARY KEY ( "customer_id" ) )
```

- This section of the statement instructs the DBMS to know that our table’s “customer_id” field will be used as the Primary Key. The Primary Key is how the data will be uniquely identified for each record set and will be essential for a healthy database relationship.

The final piece of the CREATE TABLE statement is the Semicolon. The semicolon informs the DBMS that the entire statement is complete. The “;” must be stated before any other statement can be stated or an error may occur. It is good practice to end all SQL statements with the semicolon.

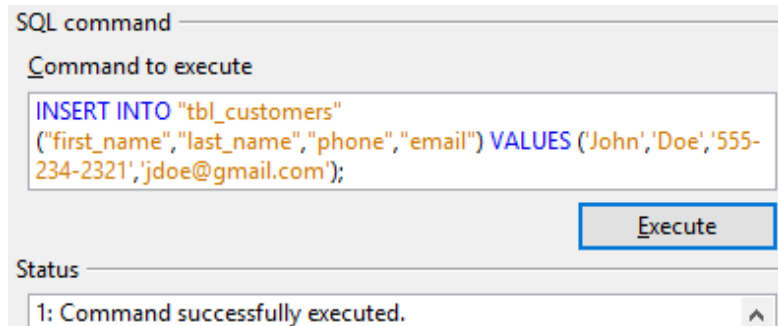
Another good habit to follow would be to always use capitalization for SQL commands in your statement. This is the recommended practice as it makes it easier for others to quickly discern the difference between SQL syntax and user data.

Using the INSERT INTO statement to add data into our database

Now that our table has been created, we need to use the INSERT INTO statement to populate the database with data. The INSERT INTO statement instructs the DBMS to that there will be new data and specifies exactly where to store that data.

Use the same SQL command window where we had used the CREATE TABLE statement and enter the following statement, then press “Execute”:

```
INSERT INTO "tbl_customers" ("first_name","last_name","phone","email") VALUES ('John','Doe','555-
234-2321','jdoe@gmail.com');
```



***NOTE** – The double quotation marks surround fieldnames while the single quotation marks are reserved for values. This is very important syntax in OpenOffice Base. If ever your code fails, check to ensure that you have not made this common syntax mistake.

INSERT INTO statement breakdown

INSERT INTO "tbl_customers"

- This particular section of the **INSERT INTO** statement informs the DBMS that there will be new data to add to the table named **"tbl_customers"**.

"customer_id", "first_name", "last_name", "phone", "email"

- This section of the statement will specify to the DBMS the exact fields to add the new data into.

VALUES 'John', 'Doe', '555-234-2321', 'jdoe@gmail.com'

- This part of the statement instructs the DBMS with the actual data for the new record set.

***NOTE** -Since we already assigned the first column **"customer_id"** as an Auto Incrementing Primary Key, it was not necessary to state it in the **INSERT INTO** statement as OpenOffice will already know what to do for Auto incrementing. If you had attempted to reference this field and implicitly add a value, you would have received an error.

Notice that the data was listed in the same order that the column names have been listed. If you had listed the data in a different order, you would have ended up with an error or improperly inserted data.

What if you don't have anything to enter into a column?

For this example, let's assume that we don't have a last name. Here is the **INSERT INTO** statement that would create a record set with missing data.

```
SQL command
Command to execute
INSERT INTO "contacts" ("first_name", "last_name", "email") VALUES
("John", NULL, "jdoe@gmail.com");
```

NULL is how you tell the DBMS that you have no data for the cell. Be careful, though. If you try to use a **NULL** in a column that must have a value, you will receive an error.

Using the UPDATE statement

Now that you have some data in your table, you may want to update it. Let's say that Mr. John Doe has a new email address that is not yet in our database. So, we will need to update John's email address so that his record is accurate.

Back in the very same SQL command window, type the following statement and **“Execute”** it:

```
UPDATE "tbl_customers" SET "email"='johndoe@rocketmail.com' WHERE "email"='jdoe@gmail.com';
```

```
SQL command
Command to execute
UPDATE "tbl_customers" SET "email"='johndoe@rocketmail.com'
WHERE "email"='jdoe@gmail.com';
Execute
Status
8: Command successfully executed.
```

UPDATE statement breakdown

UPDATE "tbl_customers"

– This part instructs the DBMS that we are going to be changing some data in the table “tbl_customers”.

SET "email"='johndoe@rocketmail.com'

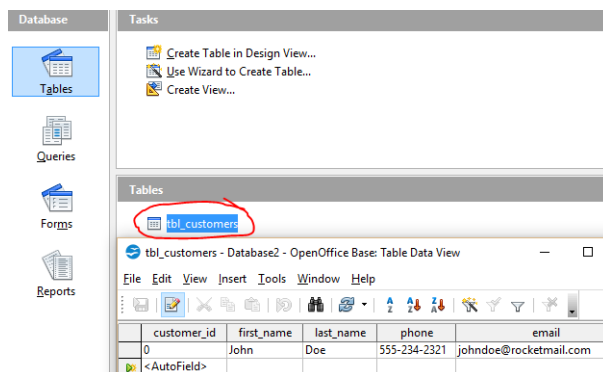
- This instructs the DBMS that there is data to change for the specified cell within the record set in this case the "email" column. You can SET as many cells of the record as you like, separating each item with a comma.

WHERE "email"='jdoe@gmail.com';

- This clause instructs the DBMS with the specific condition to be met before an update can take place. If the criterion is not met, no change should take place.

Reviewing our current changes

Back in the main window of OOB in the left menu pane, click on “Tables” then double left click on “tbl_customers” to open our table and review our current data changes.



More on SQL statements and the DBMS

These kinds of query statements make up the basic foundation to instruct a DBMS what to do and how to do it within your database. As you begin coding in more advanced programming languages, it will be absolutely essential to understand these basic SQL statements.

Many programming languages offer some sort of Graphical User Interface (GUI) that you can use to design a user front end where these SQL statements will be hidden from the user. The OOB also offers functionality to create a front-end for your database, but it is not as robust as some other programming

languages. No matter which method you choose, your code should be designed to make database manipulations seamless to the user.

What will happen if I generate an SQL error?

If there are missing characters or improper syntax, the DBMS will not execute the statement and instead, return an error. You will then need to research more about the error and work your way to a resolution. Often times OOB will not provide a very informative error and you will be forced to research and apply some “trial-and-error”. This is where practice and experience will prevail.

Should I take advantage of shortcuts I find with my DBMS?

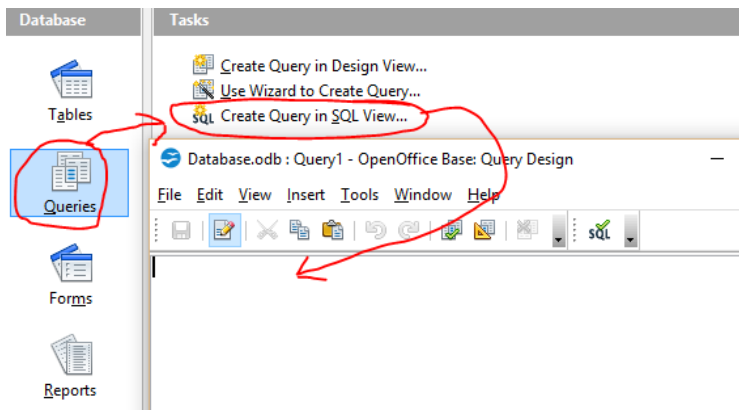
That depends, but probably not. If your DBMS has a shortcut that saves you some time typing it may be very tempting to use it. Keep in mind, though, whenever you use a shortcut that is specific to your DBMS, you may sacrifice portability.

For example you have created a database for a web based business and took advantage of some shortcuts provided with a small business DBMS that you purchased. Three years later, your business has outgrown the DBMS and now you have to invest in a more robust DBMS. You begin to move your data and code over only to find that your code no longer work with the new DBMS and you have to spend days or even weeks finding all of the places that you had used those shortcuts and make corrections.

Understanding the SELECT statement

The **SELECT** statement tells the DBMS that you want to retrieve some data out of the database. **SELECT** statements can vary from simple lists of data, to painfully complex statements.

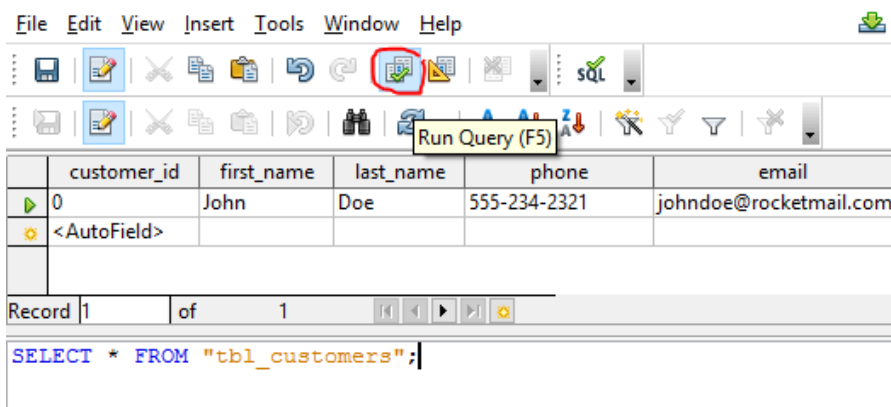
Using our newly created database, let’s retrieve all of the data from the “tbl_customers” table. Click the, “**Queries**” icon in the left menu pane then select “**Create Query in SQL View...**”



This will open a Query window where we can type in all of our **SELECT** statements and view the data.

Type in the following **SELECT** statement and click on the “Run Query” icon:

```
SELECT * FROM "tbl_customers";
```



OpenOffice Base will present you with your requested data.

The SELECT statement breakdown

SELECT *

- This instructs the DBMS that it needs to search for something. The **Asterisk *** is a wildcard symbol and means we wish to see all of the data.

FROM "tbl_customers";

- This instructs the DBMS with the correct table that it should be pulling the data from.

***NOTE - SELECT** statements can be joined together to retrieve data from several different tables. However, we will not be reviewing these more advanced statements until further in the course.

Filtering Operator Vocabulary

There are a whole host of operators that can be used to filter data. Keep in mind, each DBMS can be different and the operator syntax used can change from DBMS to DBMS.

Here are some of the most common operators that are typical within a DBMS:

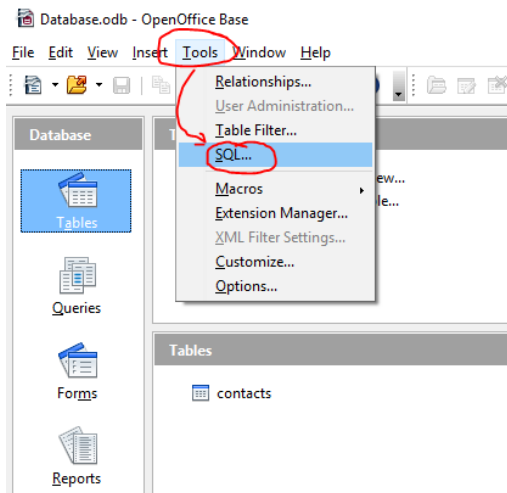
Operator Description:	
=	Compares data to your criteria to see if it is equal
<>	Compares data to your criteria to see if it is not equal
<	Compares data to your criteria to see if the data is less than your criteria
<=	Compares data to your criteria to see if the data is less than or equal to your criteria
>	Compares data to your criteria to see if the data is greater than your criteria
>=	Compares data to your criteria to see if the data is greater than or equal to your criteria
%	Denotes a wild card
IS NULL Checks the data to make sure that there is no data in the cell.	

The Structured Query Language provides you with quite a lot of flexibility in defining how to filter your data. On occasions where your statements require more than one criterion, you can use the logical operators **AND** and **OR** clauses. Both **AND** and **OR** allow you to string as many conditions together as necessary.

Filling up our table with several records of data to work from

Let's add some more records to our database so that we can perform more meaningful queries. Using the **INSERT** statement, please add a total of 9 additional records of data into "**tbl_customers**"

Remember to post your **INSERT** statements in the correct SQL command window. Close the current window and on the main menu, select "**tools**" and then "**SQL...**"



To expedite your efforts, you can chain multiple **INSERT INTO** statements together being careful to always end each statement with the Semicolon. Construct your **INSERT INTO** statements to populate the table with the following data:

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Jane', 'Doe', '555-783-1925', 'janedoe@gmail.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Tammy', 'Galeki', '785-288-1598', 'tgale23@gmail.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Bobbi', 'Dorsai', '995-711-3426', 'bobbi_j_dorsai@hotmail.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Kravel', 'Zxtul', '892-773-1124', 'zxtullery@kellogtv.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Mellissa', 'Marie', '805-102-4988', 'mm2001@gmail.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('frankie', 'Jones', '735-334-5291', 'frankmjones@gmail.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Paul', 'Saul', '225-926-3326', 'pscotts@yahoo.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Uken', 'Olsen', '523-711-9253', 'ukol00@gmail.com');
```

```
INSERT INTO "tbl_customers" ("first_name", "last_name", "phone", "email") VALUES ('Kelly', 'Brennon', '562-713-1025', 'mcfireyrabbit@gmail.com');
```

Command to execute

```

INSERT INTO "tbl_customers"
("first_name","last_name","phone","email") VALUES ('Jane','Doe','555-783-1925','janedoe@gmail.com');
INSERT INTO "tbl_customers"
("first_name","last_name","phone","email") VALUES
('Tammy','Galeki','785-288-1598','tgale23@gmail.com');
INSERT INTO "tbl_customers"
("first_name","last_name","phone","email") VALUES
('Bobbi','Dorsai','995-711-3426','bobbi_j_dorsai@hotmail.com');
INSERT INTO "tbl_customers"

```

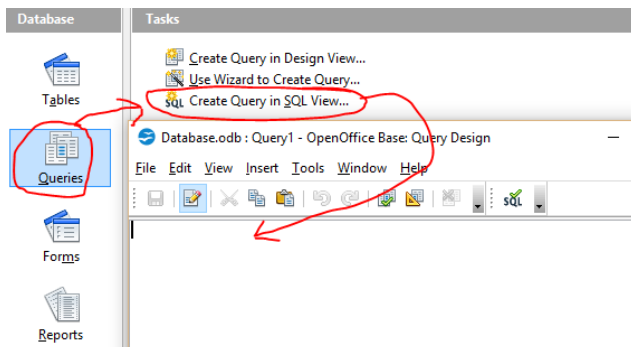
Execute

Status

1: Command successfully executed.

	customer_id	first_name	last_name	phone	email
	0	John	Doe	555-234-2321	johndoe@rocketmail.com
	1	Jane	Doe	555-783-1925	janedoe@gmail.com
	2	Tammy	Galeki	785-288-1598	tgale23@gmail.com
	3	Bobbi	Dorsai	995-711-3426	bobbi_j_dorsai@hotmail.com
	4	Kravel	Zxtul	892-773-1124	zxtullery@kellogtv.com
	5	Mellissa	Marie	805-102-4988	mm2001@gmail.com
	6	frankie	Jones	735-334-5291	frankmjones@gmail.com
	7	Paul	Saul	225-926-3326	pscotts@yahoo.com
	8	Uken	Olsen	523-711-9253	ukol00@gmail.com
	9	Kelly	Brennons	562-713-1025	mcfireyrabbit@gmail.com

Now that we have our records we can perform additional SQL statements. Return to the [SELECT](#) query window. Click on the “**Queries**” in the left menu pane and then click on “**Create Query in SQL View**”.



Type the following statement and then select the “**Run Query**” icon to execute.

Run Query

SQL

10

Record 1 of 1

```

SELECT COUNT (*) FROM "tbl_customers";

```

This **SELECT** statement employs the **COUNT** clause which instructs the DBMS to count all records of data in the “tbl_customers” table. We should have the count of **10**. The 1st record starts from index 0.

Using the LIKE clause in SELECT statements to assign wild card conditions

Now, type in this following **SELECT** statement and “Execute” it:

	customer_id	first_name	last_name	phone	email
▶ 1		Jane	Doe	555-783-1925	janedoe@gmail.com
	2	Tammy	Galeki	785-288-1598	tgale23@gmail.com
	5	Mellissa	Marie	805-102-4988	mm2001@gmail.com
	6	frankie	Jones	735-334-5291	frankmjones@gmail.com
	8	Uken	Olsen	523-711-9253	ukol00@gmail.com
	9	Kelly	Brennons	562-713-1025	mcfireyrabbit@gmail.com
Record	1	of	6		
<pre>SELECT * FROM "tbl_customers" WHERE "email" LIKE '%@gmail.com';</pre>					

In this particular query statement, we are instructing the DBMS to retrieve any records existing within the “tbl_customers” table where the email has a pattern of ‘@gmail.com’. The **LIKE** clause informs the DBMS that we will be assigning some sort of wild card or pattern to watch for. The info immediately following the percentage sign denotes what the pattern will be to watch for.

OK, now let’s make our **SELECT** statement a little more interesting and add in multiple conditions and clauses to watch for.

Once again, type the following **SELECT** statement, being very careful with the punctuation and “Execute”

```
SELECT "first_name", "phone", "email" FROM "tbl_customers" WHERE "last_name"='Olsen' AND "email" LIKE '%@gmail.com';
```

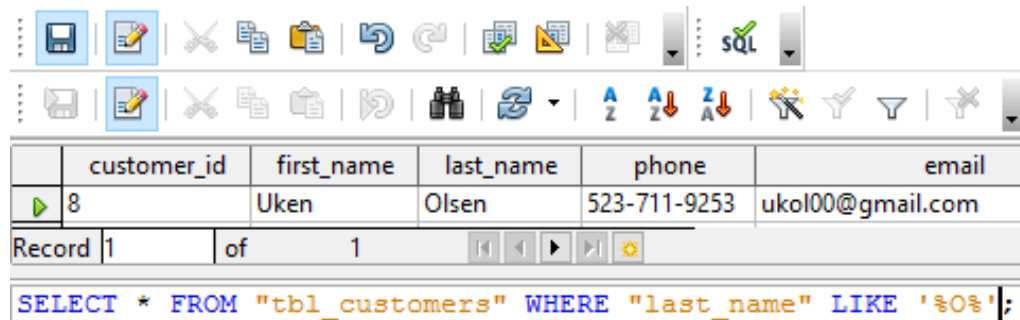
first_name	phone	email
▶ Uken	523-711-9253	ukol00@gmail.com

Record 1 of 1

```
SELECT "first_name","phone","email" FROM "tbl_customers" WHERE "last name"='Olsen' AND "email" LIKE '%@gmail.com';
```

In this **SELECT** statement we are instructing the DBMS to retrieve the first name, phone, and email from the table “tbl_customers” where the last name is Olsen and an email with a pattern of ‘@gmail.com’.

Now, let’s say that we do not know the last name’s exact spelling; we could easily have applied the following statement:



So, as you can tell, the **LIKE** clause can be extremely powerful using patterns and wild cards.

The Database schema and Database Normalization

Now that you are a little more comfortable using SQL statements, we need to complete the rest of our tables and insert more data to work with.

One thing to briefly discuss first is the **Database Schema**. The **Database Schema** is the skeletal framework for the entire database and a really good design requires considerable planning. Your database is only as good as the data you place into it. This notion holds true for how and where that data is placed. Most business databases will require vast amounts of data to be stored. A sound database schema will require all of that data to be neatly and efficiently stored into multiple related tables.

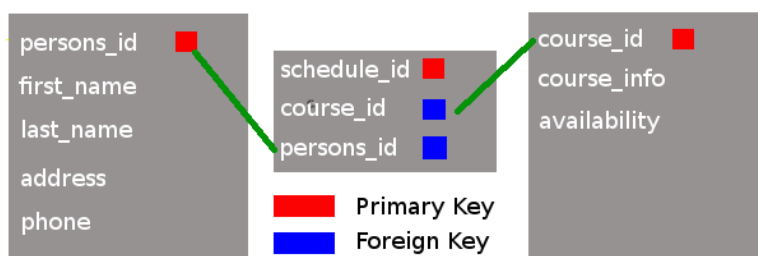
Let’s assume for a second that you have a database for a school. There of course would be students. Each of the students would possess a student id, name, contact info, grades, debt, and enrollment. Now let’s consider that there would also be Instructors. Instructors would have similar data, an id, name, contact info, assigned courses that they are teaching.

Now you would not want to have one very large and convoluted table that would contain all of this data. For one, your database would be extremely error prone. So what we require here to preserve referential integrity in our database, are several smaller tables of grouped data, where no two tables contain any repeating data. There is a term for this, it is called **Database Normalization**.

Instead of having one table for students and a separate table for instructors, we could combine the related data into just one table and call it something more generic, “persons”. The “persons” table could consist of an id, first and last name, and contact information.

We could have a second table that just contained instructor data not yet mentioned, as well as another table that was just for student data not yet mentioned. And finally, we could create a table that just contained the courses.

***NOTE** - I am absolutely positive that this data is extremely vague for a school database system, but for the sake of brevity, let’s assume that this is enough data.



This type of outline is called an **Entity Relationship Diagram (ERD)**.

In this example diagram, the three tables are interlinked via their relationships to one another. This tutorial will not be going into further detail about Referential Integrity, Database Schemas, Data Normalization, or ERD. These are all certainly great topics to research more about as building and maintaining an efficient and error free database is a very lucrative career path.

Building the tables for our database

Now that you have a better concept for database design, we are ready to build our own. It will consist of three additional tables to complement our “tbl_customers” table. (Four tables in total)

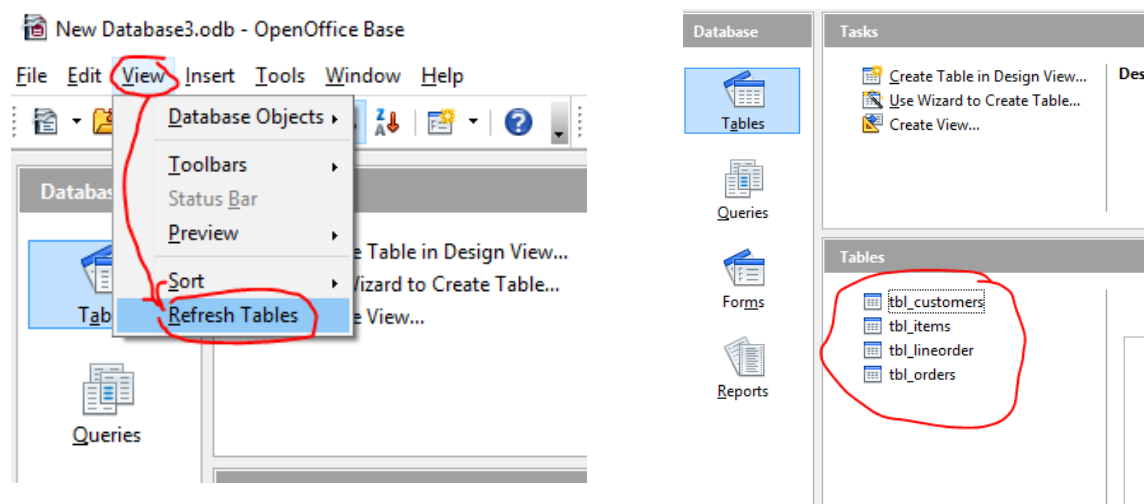
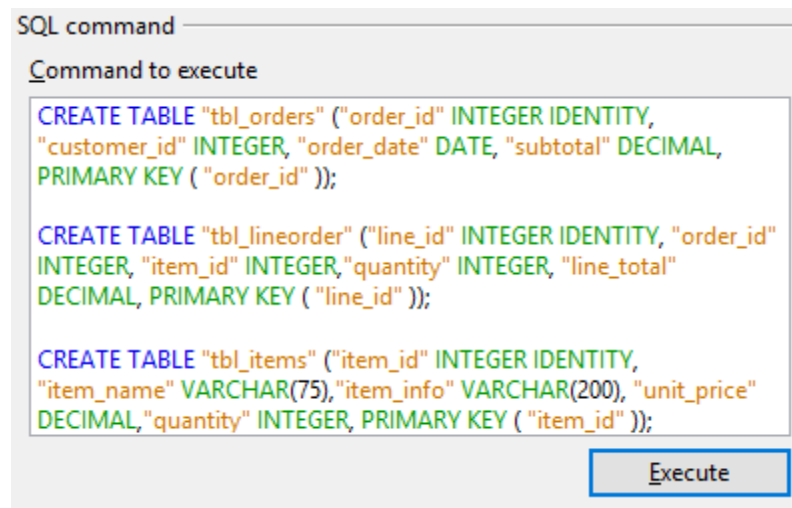
We will be employing the various SQL statements we had used earlier in this tutorial to complete and populate all of our tables with data.

Type the following commands to build each of our additional tables and **“Execute”**:

```
CREATE TABLE "tbl_orders" ("order_id" INTEGER IDENTITY, "customer_id" INTEGER, "order_date" DATE, "subtotal" DECIMAL, PRIMARY KEY ( "order_id" ));
```

```
CREATE TABLE "tbl_lineorder" ("line_id" INTEGER IDENTITY, "order_id" INTEGER, "item_id" INTEGER, "quantity" INTEGER, "line_total" DECIMAL, PRIMARY KEY ( "line_id" ));
```

```
CREATE TABLE "tbl_items" ("item_id" INTEGER IDENTITY, "item_name" VARCHAR(75), "item_info" VARCHAR(200), "unit_price" DECIMAL, "inventory" INTEGER, PRIMARY KEY ( "item_id" ));
```



***NOTE** - You may need to hit refresh for the tables to appear. Open each table and verify that the fields and Primary Keys are accurate.

Populating our database with meaningful data

Create table "tbl_orders"

Type the following statements to populate the "tbl_orders" table with the following data:

```
INSERT INTO "tbl_orders" ("order_id", "customer_id", "order_date", "subtotal") VALUES ('122200','0','2013-02-15','23.00');
```

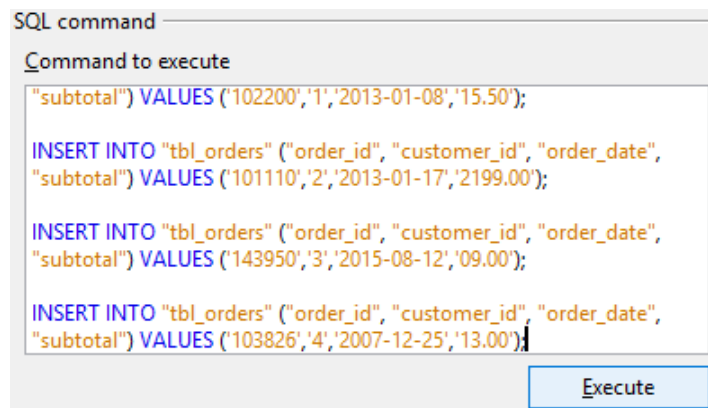
```
INSERT INTO "tbl_orders" ("order_id", "customer_id", "order_date", "subtotal") VALUES ('102200','1','2013-01-08','15.50');
```

```
INSERT INTO "tbl_orders" ("order_id", "customer_id", "order_date", "subtotal") VALUES ('101110','2','2013-01-17','2199.00');
```

```
INSERT INTO "tbl_orders" ("order_id", "customer_id", "order_date", "subtotal") VALUES ('143950','3','2015-08-12','09.00');
```

```
INSERT INTO "tbl_orders" ("order_id", "customer_id", "order_date", "subtotal") VALUES ('103826','4','2007-12-25','13.00');
```

***NOTE** - You can chain these statements to save time, but if you receive any errors, try to enter these statements one at a time and see if this will help you narrow down your syntax errors.



Your table "tbl_orders" should look like the following:

tbl_orders - Database2 - OpenO...				
File Edit View Insert Tools Window Help				
	order_id	customer_id	order_date	subtotal
	101110	2	01/17/13	2199
	102200	1	01/08/13	15.5
	103826	4	12/25/07	13
	122200	0	02/15/13	23
	143950	3	08/12/15	9
	<AutoField>			

Create table "tbl_lineorder"

Type the following statements to populate the "tbl_lineorder" table with data, being very mindful of accuracy:

```
INSERT INTO "tbl_lineorder" ("line_id", "order_id", "item_id", "quantity", "line_total") VALUES ('0', '102200', '1', '2', '31');
```

```
INSERT INTO "tbl_lineorder" ("line_id", "order_id", "item_id", "quantity", "line_total") VALUES ('1', '143950', '0', '1', '2199');
```

```
INSERT INTO "tbl_lineorder" ("line_id", "order_id", "item_id", "quantity", "line_total") VALUES ('2', '101110', '4', '2', '18');
```

```
INSERT INTO "tbl_lineorder" ("line_id", "order_id", "item_id", "quantity", "line_total") VALUES ('3', '103826', '3', '10', '230');
```

```
INSERT INTO "tbl_lineorder" ("line_id", "order_id", "item_id", "quantity", "line_total") VALUES ('4', '122200', '2', '3', '39');
```

tbl_lineorder - Database2 - OpenOffic...					
File Edit View Insert Tools Window Help					
	line_id	order_id	item_id	quantity	line_total
	0	102200	1	2	31
	1	143950	0	1	2199
	2	101110	4	2	18
	3	103826	3	10	230
	4	122200	2	3	39
	<AutoField>				
Record 1 of 5					

Create table "tbl_items"

Type the following statements to populate the “tbl_items” table with data, being very mindful of accuracy:

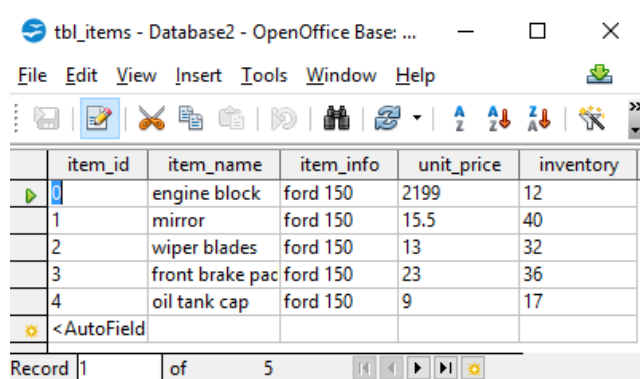
```
INSERT INTO "tbl_items" ("item_id", "item_name", "item_info", "unit_price", "inventory") VALUES ('0', 'engine block', 'ford 150', '2199', '12');
```

```
INSERT INTO "tbl_items" ("item_id", "item_name", "item_info", "unit_price", "inventory") VALUES ('1', 'mirror', 'ford 150', '15.50', '40');
```

```
INSERT INTO "tbl_items" ("item_id", "item_name", "item_info", "unit_price", "inventory") VALUES ('2', 'wiper blades', 'ford 150', '13', '32');
```

```
INSERT INTO "tbl_items" ("item_id", "item_name", "item_info", "unit_price", "inventory") VALUES ('3', 'front brake pads', 'ford 150', '23', '36');
```

```
INSERT INTO "tbl_items" ("item_id", "item_name", "item_info", "unit_price", "inventory") VALUES ('4', 'oil tank cap', 'ford 150', '9', '17');
```



The screenshot shows the OpenOffice Base application window titled "tbl_items - Database2 - OpenOffice Base: ...". The window displays a table with the following data:

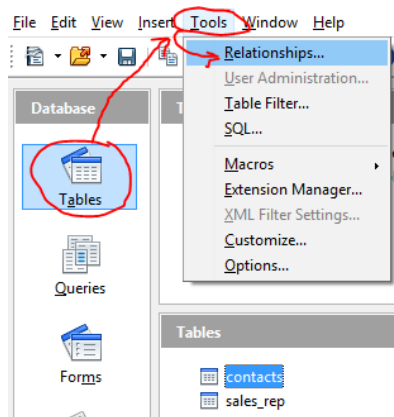
	item_id	item_name	item_info	unit_price	inventory
▶	0	engine block	ford 150	2199	12
	1	mirror	ford 150	15.5	40
	2	wiper blades	ford 150	13	32
	3	front brake pads	ford 150	23	36
	4	oil tank cap	ford 150	9	17
☼	<AutoField				

At the bottom of the window, the status bar indicates "Record 1 of 5".

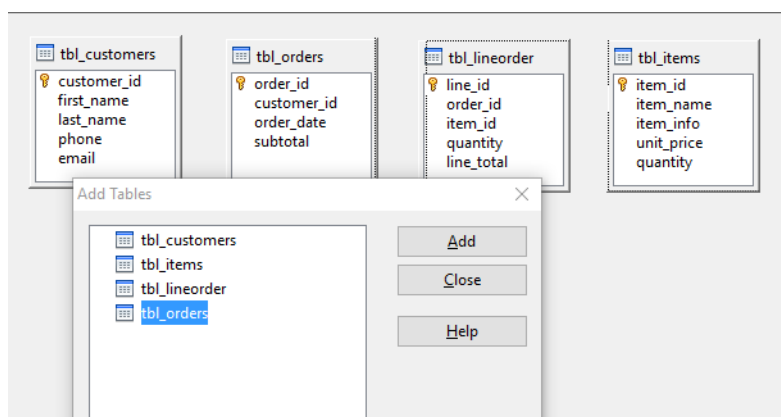
Creating relationships between our tables

Relationships are links that associate a particular field of one table to a specific field in another. In a basic summary, the relationship allows one table’s data to be available to another table. Essentially, what defines the **relationship is a link between** one table’s **Primary key** to a second table’s **Foreign key**.

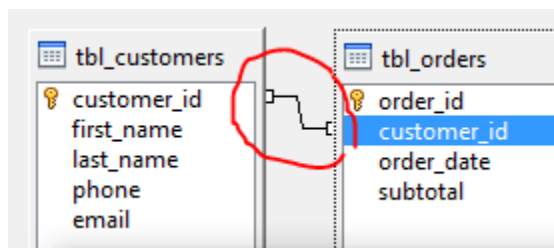
Please follow the next instructions exactly so your data will be consistent with this tutorial. Back in the main menu, click on the “Tables” icon in the left hand menu pane, in the top ribbon, select “Tools...”, and then on the drop down menu, select “Relationships...”



In the **Entity Relationship Diagram** window, select each of our four tables and click the **“Add”** button. Then arrange them all in a line in the following positions and then close the **“Add Tables”** window.



Left mouse click and hold on the Primary Key, **“customer_id”** from the **“tbl_customers”**, then drag your mouse and release over the Foreign Key, **“customers_id”** of **“tbl_orders”**. This should create a line between the two fields as in the following image:



This line denotes to our DBMS that there is a relationship between the two tables; from the Primary Key to a Foreign Key. Double click on this line.

This popup window is where you can make additional relationship configurations. Check the **“Update Cascade”** and the **“No Action”** radio options and then click on the **“OK”** button.

Relations

Tables involved

tbl_orders tbl_customers

Fields involved

tbl_orders	tbl_customers
customer_id	customer_id

Update options

☐ No action

☒ Update cascade

☐ Set null

☐ Set default

Delete options

☒ No action

☐ Delete cascade

☐ Set null

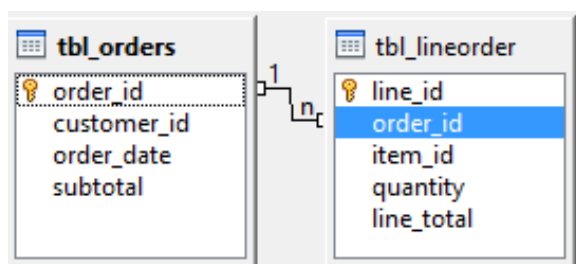
☐ Set default

OK Cancel Help

This tells the DBMS that whenever there is a change in one table, all related data in the other table should also get updated in order to maintain the data's integrity.

Now we will assign the rest of our relationships for our other tables.

From the "tbl_orders" click on "order_id" and drag your mouse over "order_id" in the "tbl_lineorder" and release the mouse. This should create the relationship link between these two tables.



Double click on this line and set the following configurations for this relationship and click, "OK".

Tables involved

tbl_lineorder tbl_orders

Fields involved

tbl_lineorder	tbl_orders
order_id	order_id

Update options

☐ No action

☒ Update cascade

☐ Set null

☐ Set default

Delete options

☒ No action

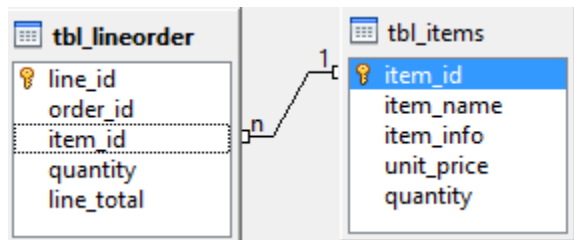
☐ Delete cascade

☐ Set null

☐ Set default

OK Cancel Help

Again, click and hold your mouse over “**item_id**” from the “**tbl_lineorder**” and drag your mouse over “**item_id**” in the “**tbl_items**” and release the mouse to create their relationship.



Click on this line and ensure the correct configurations settings and click the “**OK**” button.

Tables involved

tbl_lineorder tbl_items

Fields involved

tbl_lineorder	tbl_items
item_id	item_id

Update options

☐ No action

☒ Update cascade

☐ Set null

☐ Set default

Delete options

☒ No action

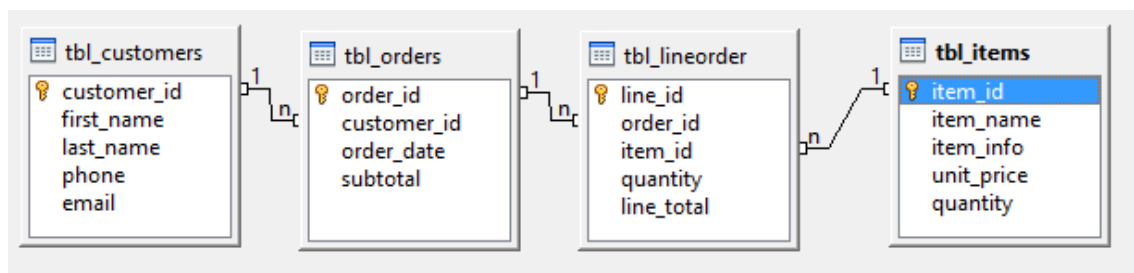
☐ Delete cascade

☐ Set null

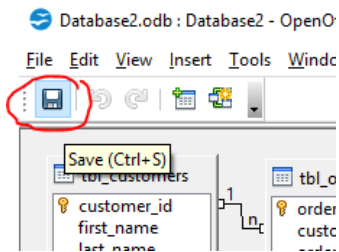
☐ Set default

OK Cancel Help

Our final ERD should appear like the following:



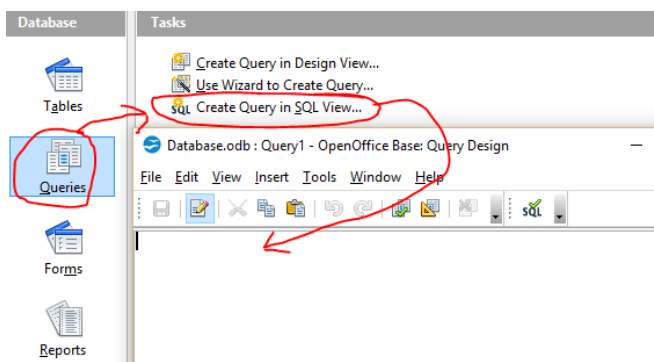
From the “**Relationship**” window, select the “**Save**” icon to save our relationship and settings.



Performing more advanced query statements

Now that we have an established database structure with relational constraints and meaningful data, we will now try out some more advanced kinds of query statements.

From the main window in OpenOffice Base, click on the “**Queries**” icon in the left hand menu pane and select “**Create Query in SQL View...**”



In the Query Design window, type in the following statement and “**Execute**” it.

```
SELECT "first_name" AS "First Name:", "last_name" AS "Last Name:", "phone" AS "Phone Number:" FROM
"tbl_customers" WHERE "first_name"='Jane';
```

	First Name:	Last Name:	Phone Number:
▶	Jane	Doe	555-783-1925
Record 1 of 1			

```
SELECT "first_name" AS "First Name:", "last_name" AS "Last Name:", "phone" AS "Phone Number:"
FROM "tbl_customers" WHERE "first_name"='Jane';
```

What we have just done was to tell the DBMS what data to gather and how exactly to display that data, substituting alias field names instead of our original ones that were lowercased and contained

underscores. So instead of **first_name**, **last_name**, and **phone**, we now see the more presentable version, **First Name:**, **Last Name:** and **Phone Number:**.

Now, type in the following statement and “Execute” it.

```
SELECT * FROM tbl_customers INNER JOIN tbl_orders ON  
tbl_customers.customer_id=tbl_orders.customer_id;
```

	customer_id	first_name	last_name	phone	email	order_id	customer_id	order_date	subtotal
▶ 0		John	Doe	555-234-2321	johndoe@rocketma	122200	0	02/15/13	23
1		Jane	Doe	555-783-1925	janedoe@gmail.com	102200	1	01/08/13	15.5
2		Tammy	Galeki	785-288-1598	tgale23@gmail.com	101110	2	01/17/13	2199
3		Bobbi	Dorsai	995-711-3426	bobbi_j_dorsai@hotmail	143950	3	08/12/15	9
4		Kravel	Zxtul	892-773-1124	zxtullery@kellogtv.c	103826	4	12/25/07	13
☀ <AutoField>						<AutoField>			
Record 1	of 5								

```
SELECT * FROM tbl_customers INNER JOIN tbl_orders ON  
tbl_customers.customer_id=tbl_orders.customer_id;
```

With this particular statement we were able to instruct the DBMS to retrieve data from two of our tables by using the clause **INNER JOIN** in conjunction with the **ON** clause.

Let’s take a moment to explain how the **INNER JOIN** syntax works as **JOIN** statements can be some of the more confusing types of query statements.

The **JOIN** statement worked because we instructed the DBMS to **SELECT** all fields from our “customers” table, then we informed the DBMS that we intended to also include an additional table with the inclusion of the **INNER JOIN** clause followed immediately with the table, “orders”. Finally, our statement uses the **ON** clause, informing the DBMS how the two tables are related.

When it comes to the **JOIN** statements, there are several different types to choose from, such as the **INNER**, **OUTER**, **LEFT**, and the **RIGHT** Joins. I strongly encourage you to do some additional research about the differences between the [SQL JOIN statements](#) and when they would best apply.

Type in the following statement and “Execute”

```
SELECT COUNT(order_id) AS "TRANSACTIONS:", SUM(subtotal) AS "NET INCOME:" FROM tbl_orders;
```


	TRANSACTIONS:	NET INCOME:	
▶	5	2259.5	
Record	1	of	1
SELECT COUNT (order_id) AS "TRANSACTIONS:", SUM(subtotal) AS "NET INCOME:" FROM tbl_orders;			

Here we have made use of a couple of SQL math functions “**COUNT ()**” and “**SUM ()**”, then presented the data neatly with the alias clause “**AS**”.

Now we will try one more **SELECT** statement and we will really make this interesting and complicated. Type in the following statement and “**Execute**” it:

```
SELECT DISTINCT(tbl_orders.order_id), tbl_orders.subtotal, tbl_customers.first_name,
tbl_customers.last_name FROM tbl_orders, tbl_customers INNER JOIN tbl_items ON
(tbl_customers.customer_id = tbl_orders.customer_id) WHERE order_date BETWEEN '2013-01-08' AND
'2016-01-01' ORDER BY tbl_customers.first_name DESC;
```

	order_id	subtotal	first_name	last_name	
▶	101110	2199	Tammy	Galeki	
	122200	23	John	Doe	
	102200	15.5	Jane	Doe	
	143950	9	Bobbi	Dorsai	
Record	1	of	4		
SELECT DISTINCT (tbl_orders.order_id), tbl_orders.subtotal, tbl_customers.first_name, tbl_customers.last_name FROM tbl_orders, tbl_customers INNER JOIN tbl_items ON (tbl_customers.customer_id = tbl_orders.customer_id) WHERE order_date BETWEEN '2013-01-08' AND '2016-01-01' ORDER BY tbl_customers.first_name DESC;					

We have just instructed the DBMS to retrieve only the unique order id and aggregate the subtotal from the orders table. We’ve also instructed the DBMS to retrieve the first name and last name from the customers table that have order dates that fall between two specific date ranges, and display that information to the screen in a specific order, descending down, reverse alphabetically.

Another way to make your **JOIN** statements easier to construct is by defining table aliases. Please review the following example:

	first_name	last_name	order_id	subtotal
▶	Tammy	Galeki	101110	2199
	John	Doe	122200	23
	Jane	Doe	102200	15.5
	Bobbi	Dorsai	143950	9

Record 1 of 4

```

SELECT a.first_name, a.last_name, b.order_id, b.subtotal
FROM tbl_customers a INNER JOIN
tbl_orders b ON a.customer_id = b.customer_id
WHERE order_date BETWEEN '2013-01-08' AND '2016-01-01'
ORDER BY a.first_name DESC;

```

In the **FROM** clause, I defined the **tbl_customers** as **a**. Then in the **INNER JOIN** clause, I defined the **tbl_orders** as **b**. This way I never again have to write out **tbl_customers** or **tbl_orders** in future statements. This makes the statements easier to write and to follow. This process is very crucial for constructing complex SQL statements.

***NOTE** - There is a bug in OpenOffice where using the alias of **d** will produce an error as it appears to be an undocumented, reserved keyword. I recommend that if you need to use the alias **d**, to enclose it in quotes like so, **"d"**.

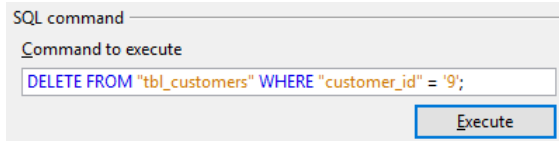
Also, **JOINS** can be chained together in one statement. In most real-world databases, multiple **JOIN** queries are essential as databases contain enormous amounts of data stored in a plethora of tables. Without a well constructed query, that data would be absolutely meaningless.

Deleting data from databases

SQL has syntax for deleting data, but treat this particular command with respect. The bane of any database is garbage data and data loss. So, in practice, delete operations are generally not performed manually, but are likely generated via a front end system which will maintain safe-guards of data and provide the user with warnings.

Type in the following command and then **"Execute"**.

```
DELETE FROM "tbl_customers" WHERE "customer_id" = '9';
```

A screenshot of an SQL command window. The title bar says "SQL command". Below it, there is a label "Command to execute" and a text input field containing the SQL command: `DELETE FROM "tbl_customers" WHERE "customer_id" = '9';`. To the right of the input field is a button labeled "Execute".

```
SQL command
Command to execute
DELETE FROM "tbl_customers" WHERE "customer_id" = '9';
Execute
```

This will remove the entire record of data from the customer's table where the customer's id was 9.

The final query request:

OK, it is time to build customer invoice statements. To do this, we will need to query data from all four of our tables. For your final query, see if you can write out your own **SELECT** statement to query our database for all of the necessary data.

***NOTE** - You will need to send a screen shot of your final query and the returned results to an instructor for a pass on this drill.

Your statement needs to query our database for the following:

- The order id
- The order date
- The customer's first name
- The customer's last name
- The item name
- The quantity
- The unit price
- The line total
- Your query statement needs to present this data ordered by the order date.

Be sure that your data is presented in the following arrangement:

	order_id	order_date	first_name	last_name	item_name	quantity	unit_price	line_total
--	----------	------------	------------	-----------	-----------	----------	------------	------------

It is highly likely that you will get stuck trying to complete this drill. It is anticipated that you will need to research a little more about SQL. Programmers oftentimes are forced to research, plan, and apply trial and error in order to satisfy their client/employer's requests. So there is no reason to feel bad at all.

Don't get discouraged by this assignment. Be sure to **"think like a programmer"** and break down your code into little manageable pieces and see if you can get the smaller pieces to work before moving on. Take each hurdle one by one until you can finally tie it all together and resolve the greater problem.