SQL for Beginners: Part 3 - Database Relationships

<https://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561>

by [Burak Guzel](https://tutsplus.com/authors/burak-guzel)13 Jan 2010

Length:LongLanguages:

[Databases](https://code.tutsplus.com/categories/databases)[Web Development](https://code.tutsplus.com/categories/web-development)[Back-End](https://code.tutsplus.com/categories/back-end)[MySQL](https://code.tutsplus.com/categories/mysql)[Relational Databases](https://code.tutsplus.com/categories/relational-databases)

Today, we continue our journey into the world of SQL and relational database systems. In this part three of the series, we'll learn **how to work with multiple tables** that have relationships with each other. First, we will go over some core concepts, and then **will begin working with JOIN queries in SQL**.

You can also see SQL databases in action by checking out the [SQL scripts, apps and add-ons](https://codecanyon.net/tags/sql) on Envato Market.

**Catch Up**

* [SQL for Beginners: Part 1](http://net.tutsplus.com/tutorials/other/sql-for-beginners/)
* [SQL for Beginners: Part 2](http://net.tutsplus.com/tutorials/other/sql-for-beginners-part-2/)

**Introduction**

When creating a database, common sense dictates that we use separate tables for different types of entities. Some examples are: customers, orders, items, messages etc... But we also need to have relationships between these tables. For instance, customers make orders, and orders contain items. These relationships need to be represented in the database. Also, when fetching data with SQL, we need to use certain types of JOIN queries to get what we need.

**There are several types of database relationships**. Today we are going to cover the following:

* **One to One Relationships**
* **One to Many and Many to One Relationships**
* **Many to Many Relationships**
* **Self Referencing Relationships**

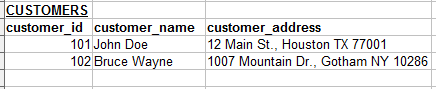
When selecting data from multiple tables with relationships, we will be using the JOIN query. There are several types of JOIN's, and we are going to learn about the the following:

* **Cross Joins**
* **Natural Joins**
* **Inner Joins**
* **Left (Outer) Joins**
* **Right (Outer) Joins**

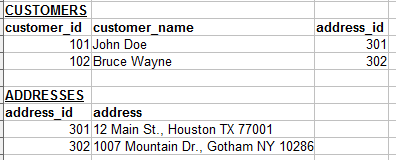
We will also learn about the **ON clause** and the **USING clause**.

**One to One Relationships**

Let's say you have a table for customers:



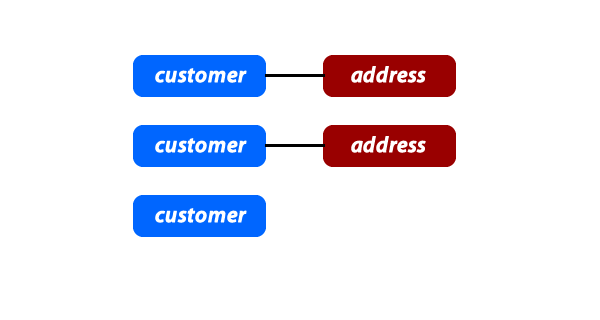
We can put the customer address information on a separate table:



Now we have a relationship between the **Customers table** and the **Addresses table**. If each address can belong to only one customer, this relationship is "One to One". Keep in mind that this kind of relationship is not very common. Our initial table that included the address along with the customer could have worked fine in most cases.

Notice that now there is a field named "address\_id" in the Customers table, that refers to the matching record in the Address table. This is called a "Foreign Key" and it is used for all kinds of database relationships. We will cover this subject later in the article.

We can visualize the relationship between the customer and address records like this:



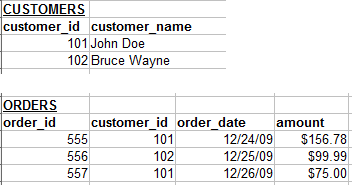
Note that the existence of a relationship can be optional, like having a customer record that has no related address record.

**One to Many and Many to One Relationships**

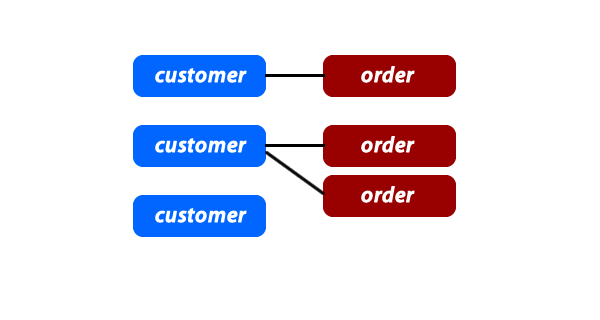
This is the most commonly used type of relationship. Consider an e-commerce website, with the following:

* Customers can make many orders.
* Orders can contain many items.
* Items can have descriptions in many languages.

In these cases we would need to create "One to Many" relationships. Here is an example:



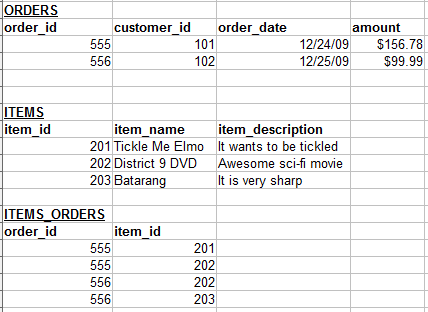
Each customer may have zero, one or multiple orders. But an order can belong to only one customer.



**Many to Many Relationships**

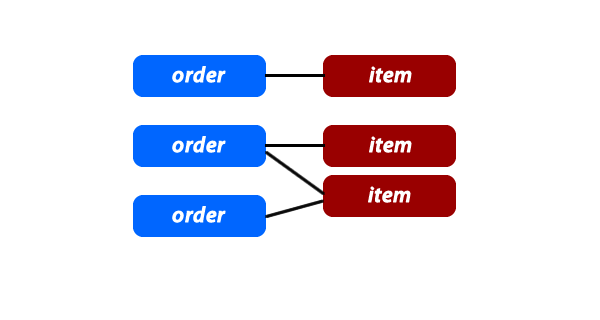
In some cases, you may need multiple instances on both sides of the relationship. For example, each order can contain multiple items. And each item can also be in multiple orders.

For these relationships, we need to create an extra table:

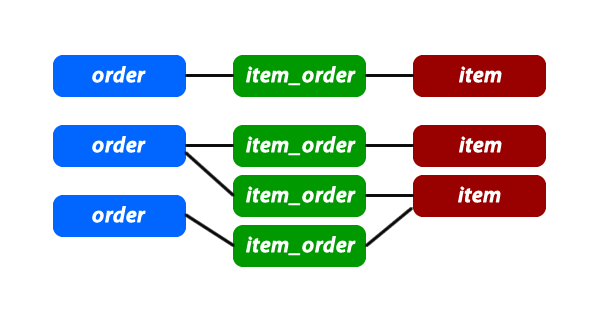


The Items\_Orders table has only one purpose, and that is to create a "Many to Many" relationship between the items and the orders.

Here is a how we can visualize this kind of relationship:

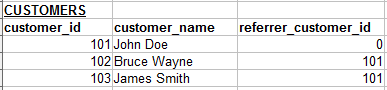


If you want to include the items\_orders records in the graph, it may look like this:



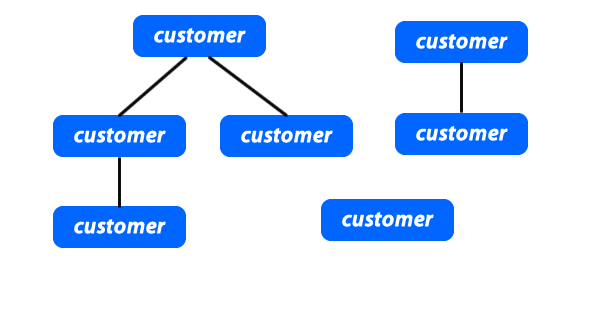
**Self Referencing Relationships**

This is used when a table needs to have a relationship with itself. For example, let's say you have a referral program. Customers can refer other customers to your shopping website. The table may look like this:



Customers 102 and 103 were referred by the customer 101.

**This actually can also be similar to "one to many" relationship since one customer can refer multiple customers. Also it can be visualized like a tree structure:**



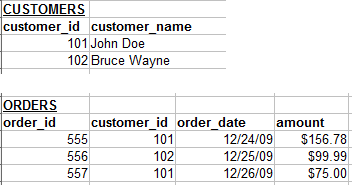
One customer might refer zero, one or multiple customers. Each customer can be referred by only one customer, or none at all.

If you would like to create a self referencing "many to many" relationship, you would need an extra table like just like we talked about in the last section.

**Foreign Keys**

So far we have only learned about some of the concepts. Now it is time to bring them to life using SQL. For this part, we need to understand what Foreign Keys are.

In the relationship examples above, we always had these "\*\*\*\*\_id" fields that referenced a column in another table. In this example, the customer\_id column in the Orders table is a Foreign Key column:



With a database like MySQL, there are two ways to create foreign keys columns:

**Defining the Foreign Key Explicitly**

Let's create a simple customers table:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4** | **CREATE TABLE customers (**  **customer\_id INT AUTO\_INCREMENT PRIMARY KEY,**  **customer\_name VARCHAR(100)**  **);** |

Now the orders table, which will contain a Foreign Key:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **CREATE TABLE orders (**  **order\_id INT AUTO\_INCREMENT PRIMARY KEY,**  **customer\_id INT,**  **amount DOUBLE,**  **FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)**  **);** |

Both columns (**customers.customer\_id** and **orders.customer\_id**) should be the same exact data structure. If one is INT, the other one should not be BIGINT for example.

***Please note that in MySQL only the InnoDB engine has full support for Foreign Keys. But other storage engines will still allow you to specify them without giving any errors. Also the Foreign Key column is indexed automatically, unless you specify another index for it.***

**Without Explicit Declaration**

Same orders table can be created without explicitly declaring the customer\_id column to be a Foreign Key:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **CREATE TABLE orders (**  **order\_id INT AUTO\_INCREMENT PRIMARY KEY,**  **customer\_id INT,**  **amount DOUBLE,**  **INDEX (customer\_id)**  **);** |

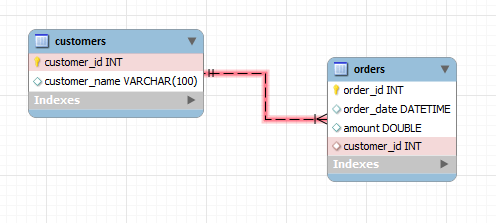
When retrieving data with a JOIN query, you can still treat this column as a Foreign Key even though the database engine is not aware of that relationship.

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM orders  JOIN customers USING(customer\_id) |

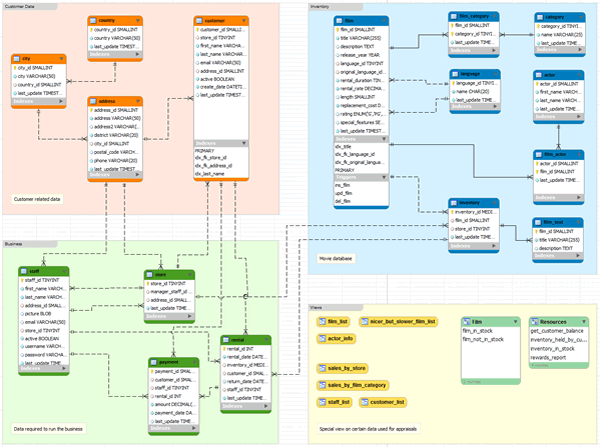
We are going to learn about JOIN queries further in the article.

**Visualizing the Relationships**

My current favorite software for designing databases and visualizing the Foreign Key relationships is [MySQL Workbench](http://wb.mysql.com/).



Once you design your database, you can export the SQL and run it on your server. This comes in very handy for bigger and more complex database designs.



**JOIN Queries**

To retrieve data from a database that has relationships, we often need to use JOIN queries.

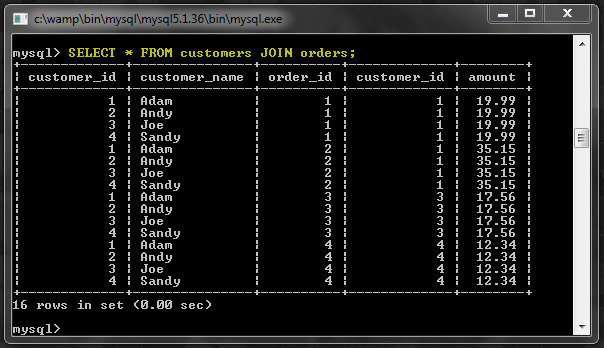
**Before we get started, let's create the tables and some sample data to work with.**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | CREATE TABLE customers (      customer\_id INT AUTO\_INCREMENT PRIMARY KEY,      customer\_name VARCHAR(100)  );    CREATE TABLE orders (      order\_id INT AUTO\_INCREMENT PRIMARY KEY,      customer\_id INT,      amount DOUBLE,      FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)  );    INSERT INTO `customers` (`customer\_id`, `customer\_name`) VALUES  (1, 'Adam'),  (2, 'Andy'),  (3, 'Joe'),  (4, 'Sandy');    INSERT INTO `orders` (`order\_id`, `customer\_id`, `amount`) VALUES  (1, 1, 19.99),  (2, 1, 35.15),  (3, 3, 17.56),  (4, 4, 12.34); |

We have 4 customers. One customer has two orders, two customers have one order each, and one customer has no order. Now let's see the different kinds of JOIN queries we can run on these tables.

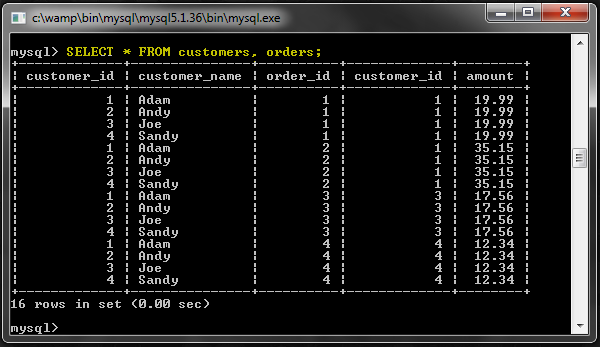
**Cross Join**

This is the default type of JOIN query when no condition is specified.



The result is a so called "**Cartesian product**" of the tables. It means that each row from the first table is matched with each row of the second table. Since each table had 4 rows, we ended up getting a result of 16 rows.

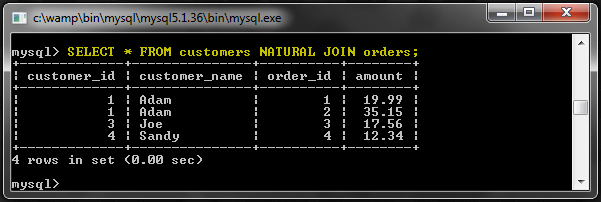
**The JOIN keyword can be optionally replaced with a comma instead.**



Of course this kind of result is usually not useful. So let's look the other join types.

**Natural Join**

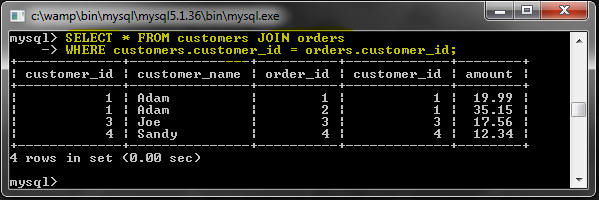
**With this kind of JOIN query, the tables need to have a matching column name**. In our case, both the tables have the customer\_id column. So, MySQL will join the records only when the value of this column is matching on two records.



As you can see the **customer\_id** column is only displayed once this time, because the database engine treats this as the **common column**. We can see the two orders placed by Adam, and the other two orders by Joe and Sandy. Finally we are getting some useful information.

**Inner Join**

**When a join condition is specified, an Inner Join is performed**. In this case, it would be a good idea to have the **customer\_id** field match on both tables. The results should be similar to the **Natural Join**.



The results are the same except a small difference. The **customer\_id** column is repeated twice, once for each table. The reason is, we merely asked the database to match the values on these two columns. But it is actually unaware that they represent the same information.

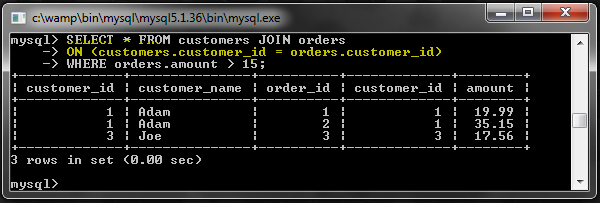
**Let's add some more conditions to the query.**



This time we received only the orders over $15.

**ON Clause**

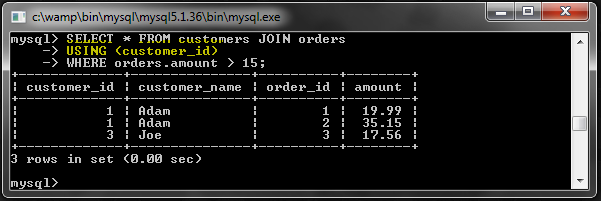
Before moving on to other join types, we need to look at the **ON clause.** This is useful for putting the JOIN conditions in a separate clause.



Now we can distinguish the JOIN condition from the WHERE clause conditions. But there is also a slight difference in functionality. We will see that in the LEFT JOIN examples.

**USING Clause**

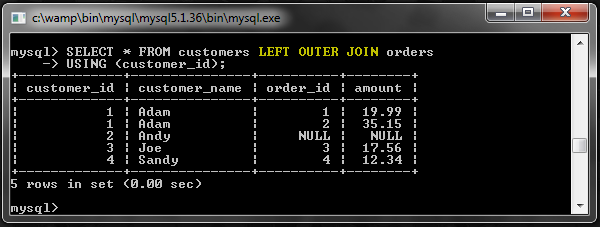
USING clause is similar to the **ON clause**, but it's shorter. If a column is the same name on both tables, we can specify it here.



In fact, this is much like the NATURAL JOIN, so the join column (customer\_id) is not repeated twice in the results.

**Left (Outer) Join**

**A LEFT JOIN** is a type of **Outer Join**. In these queries, if there is no match found from the second table, the record from the first table is still displayed.



Even though Andy has no orders, his record is still being displayed. The values under the columns of the second table are set to NULL.

**This is also useful for finding records that do not have relationships. For example, we can search for customers who have not placed any orders.**

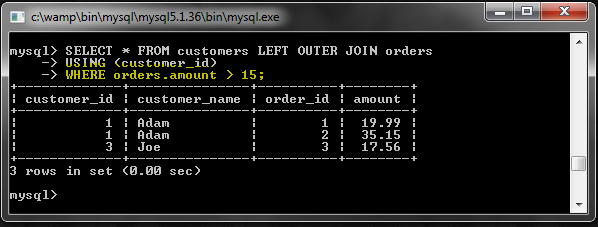


All we did was to look for NULL values for the **order\_id.**

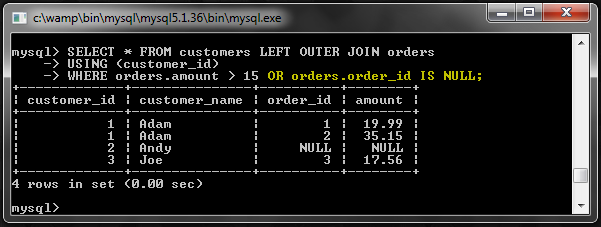
Also note that the OUTER keyword is optional. You can just use LEFT JOIN instead of LEFT OUTER JOIN.

**Conditionals**

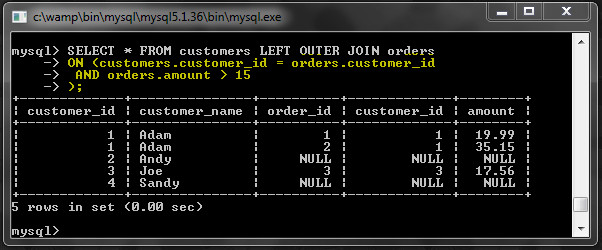
Now let's look at a query with a condition.



So what happened to Andy and Sandy? LEFT JOIN was supposed to return customers with no matching orders. The problem is that the WHERE clause is blocking those results. To get them we can try to include the NULL condition as well.



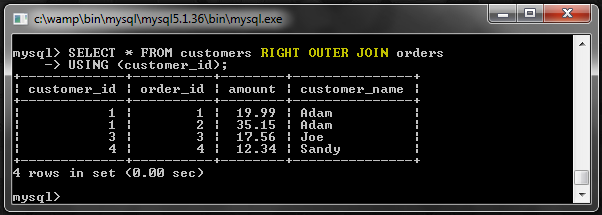
We got Andy but no Sandy. Still this does not look right. To get what we want, we need to use the ON clause.



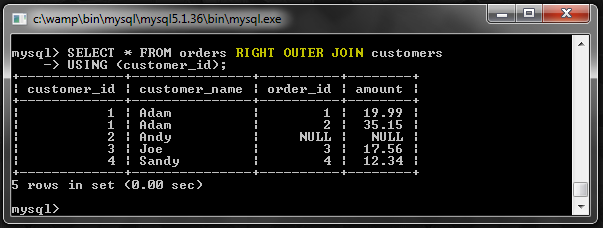
Now we got everyone, and all orders above $15. As I said earlier, the **ON clause** sometimes has slightly different functionality than the **WHERE clause**. In an Outer Join like this one, rows are included even if they do not match the ON clause conditions.

**Right (Outer) Join**

A RIGHT OUTER JOIN works exactly the same, but the order of the tables are reversed.



This time we have no NULL results because every order has a matching customer record. We can change the order of the tables and get the same results as we did from the LEFT OUTER JOIN.



**Now we have those NULL values *because the customers* table is on the right side of the join.**

**Conclusion**

Thank you for reading the article. I hope you enjoyed it! Please leave your comments and questions, and have a great day!