Joins

In order for data to be retrieved properly from a relational database we have learned that the tables have to be normalized and the data is usually split up between multiple tables. But when that data needs to be retrieved again for the user, it needs to make sense and many times that means joining the data back together again before it is presented as output (or useful information) to the user.

For example, if we have tables in our database that we have separated out during our design they might look something like this. All the artists are their own entity and all the artwork they create is it's own entity. This is how we had to normalize the database for the relations to make sense so there were not redundancies and the data could be retrieved properly. With these two tables the artist_id is how they are related, which is the primary key of the artist table, and the column by the same name artist_id in the Artwork table where is the foreign key. The artist and artwork table relationship is a one-to-many. One artist can create many pieces of art. And one artwork is assumed to have only one artist. Therefore the artist_id can be repeated multiple times as a foreign key in the artwork table, but is unique only once in the artist table. For example Leonardo da Vinci has the artist_id of 13 and he has 3 different artworks associate with him in the Artwork table. The Mona Lisa, The Last Supper and Head of a Woman.

But when our user wants to get information about all the artwork created by Leonardo Da Vinci they won't know that Leonardo's artist_id is number 13 in our database. So the user won't know to ask for all artwork information from artist number 13. They just want to see all of Leonardo's artwork. So in this case they have a drop down list that has all the artists. Or maybe they enter the name into an input box, whatever it might be, their selection can become a part of the WHERE clause to filter the results.

This brings up the artwork that has to do with that artist. But the field that was used in the WHERE clause, the artist's name, and the artwork that comes up by the file name to show the results are in two different tables.

And if the user clicks for more information on one of the artworks, you can see even more attributes or columns from the artwork table. But the way the user got the results is by using an artist name from the artist table. We have to join these two tables together to use columns from both tables in this type of a query.

The default join (and most common type of join) is the inner join. The inner join combines two tables and returns only matching rows. With Inner joins, each row that shows up in the result set must have matching column values in the different tables. The inner join query compares each row one table with each row of another table to find all pairs of rows that satisfy the join. So all primary keys of one table for example match up to all foreign keys in another table.

Notice in the SELECT statement we have column names that belong to two different tables. We will be adding a few new clauses to our query to make sure we are getting data properly from both tables. When we state what table we are getting data from we have to list both tables. So we list one table (in our case the artist table) and then the keyword JOIN and the second table. When we just use the keyword JOIN it is assumed that it is an INNER JOIN. You could use those two keywords together and it would mean the same thing. Next we have to tell exactly how those tables are related. What key is it that relates (or links) those tables together? It's the artist_id column right? That's the key that is the same for both tables, it's the primary key for the artist table and the foreign key for the artwork table. This is the field that they both have in common. Our query has to know this. So we add the ON clause that just reaffirms how those tables are related; the artist_ id in one and the artist_id in the other. Since we've named them exactly the same name (artist_id) we have to tell the query which table we are talking about. We have use the *qualified column name*, which means we just proceed the column name with the table name and a period. Now it knows which is which. So we are saying here: we are getting information or columns from two different tables, from the artist table and the artwork table. They are both related by the key artist_id. The artist_id of the artist table and the artist_id of the artwork table. Because we also have a WHERE clause filtering it down to only first name 'Leonardo', the result set below gets is only the artwork that fits in the artist_id of Leonardo Da Vinci.

Here we have 3 tables that are joined. We have columns in the SELECT statement from 3 different tables. We are joining the artwork table, the keybridge table and the keyword table. For each join we have the ON clause showing how the two tables are related. Here the user typed in the keyword 'flower' and that became the filter value in the WHERE clause. Then each table was joined so that the artwork showed up and when they get more details about that artwork they can see all the information from each of the column names we have in the SELECT statement.

So again only the rows that had a matching primary and foreign key relationship will show up in the results with an inner join.

OUTER JOINS are not as common as INNER JOINS. Outer joins return every single row of one table and the matching rows of another table.

Say we have an artist that is in our artist table but he doesn't have any artwork associated with him in the related artwork table. But we want to list all the artists with their associated artwork and we want to make sure we included all artists even if they didn't have artwork associate with them. This would be where we would use a OUTER JOIN. LEFT OUTER JOIN and RIGHT OUTER JOIN mean the same thing as LEFT JOIN and RIGHT JOIN. We just make sure the artist table is on the left in this case because with a left join the table that you want to show all the rows with

should be on the left. With a right join the table you want to show all the rows for will be on the right. Usually you just use left and make sure the table you want all the information from is first (or on the left).

So here we have the first and last name of the artist and the title of the artwork and the location of the artwork; columns from both tables. With the left (outer implied) join then it will get every row from the artist table (it's listed on the left) and only the associated artwork from the right table, artwork. You can see that Michelangelo is an artist without any artwork entered into our database yet. See how his first and last name show up in the result set with nulls filling the other columns since he doesn't have any artwork associate with him in the artwork table.

Just for comparison you can see that if we'd done an inner join here the conditions of the join (meaning the primary and foreign key relation) that the join was created on would have to be true for the result set to show that row. Since Michelangelo doesn't not have that relationship with the two tables yet, he does not show up. Michelangelo has no foreign key in the artwork table that is related to his primary key in the artist table.

The Venn Diagram shows that the table we had on the left (in our case the artist) will have all rows show up and the table on the right (artwork) has only those that have information associated with the join will show up.

The last type of JOIN is a FULL OUTER JOIN this is when the result set includes rows from both tables, even if there is not a related key. This type of join is even more uncommon than the Left and Right joins. MySQL doesn't have syntax to run a full outer join so you can use unions that can simulate it. Unions will remove duplicates so if you want a full outer join you can use the ALL keyword. Again this is very uncommon and would rarely be used, but it's good to know it is available incase an exception report is needed that shows exceptions to the data for some reason. There isn't a good, logical reason to do this with the artist and artwork example and since SQL allows for this syntax but MySQL doesn't I am showing an example from W3schools here.

This is a customer table and a related orders table. There is a customer id that related the tables.

When a full outer join is run the all customers show up (even if they didn't make an order) and all orders show up (even if they aren't related to a customer).

Here is the syntax we would use to simulate a FULL OUTER JOIN. Notice how it uses the UNION keyword to do this.

There are other types of joins like Cross joins but they are beyond the scope of this course. Most likely when you join tables it will be an inner join, which uses just the JOIN keyword.