Lesson 5-Selecting Data from Multiple Tables (JOINs)

Lesson 5 Concepts

1. MySQL is a *Relational* Database Management System
2. Relationships between tables are established with Primary keys and Foreign keys
3. A query can collect a result set from across multiple tables by joining the tables together
4. Queries are more easily readable when using table aliases

MySQL is a Relational Database Management System

The main benefit of a *relational* database management system (RDBMS) is that it has the ability to make connections between multiple tables and use these relationships to generate a result set from across multiple tables. Imagine a library with a table of books and a table of members. The main relationship between these two tables would be: which books were borrowed by which members? With a third table called, “loans”, a relational database would manage these relationships without having to create a row with all of the book’s information, and all of the member’s information for each borrowed book.

Relationships between Tables are established with Primary Keys and Foreign Keys

You may have noticed that each table that we have used has its first column as ‘id’. This gives each row in the table a unique identifier to distinguish it from the other rows. The first column with the unique identifier is referred to as the “primary key” for the table. Referencing this primary key in other tables is how we build relationships between them. From our library example above, each row in the loans table would refer to the book and its borrower by their primary key values, rather than by book title and name of borrower. When a column tracks a primary key from another table, it is referred to as a “foreign key”. For example, the id column in the books table is the primary key for that table; if the loans table had a book\_id column that referenced the primary key from the books table, it would be a foreign key column. By identifying the relationships between the keys in our queries, we are able to join tables together in our results.

A Query Can Collect a Result Set From Across Multiple Tables by Joining the Tables Together

To create a result set from multiple tables, we add the **JOIN** and **ON** clauses to the **FROM** clause of our query. The pattern is as follows:

SELECT \* FROM ***first\_table* JOIN *second\_table* ON *first\_table.foreignkey* = *second\_table.primarykey***;

In the above, we **JOIN** the *names* of the two tables we want to join, then specify the primary key / foreign key relationship between the two tables in the ON clause with the = and the names of the columns from the tables that contain the primary or foreign keys. You can think of this as a **WHERE** clause: …WHERE *first\_table.foreignkey* = *second\_table.primarykey*; Note the period (with no spaces) between the table name and the key column. This is “dot notation” and you can think of it as a possessive apostrophe: books.id is saying, books**’** id (the id column belonging to the books table).

Using our Pet Store tables, we **JOIN** the *sales* table with the *employees* table, specifying that we want the rows where the *employee* column (foreign key) of the sales table matches the *id* column (primary key) of the employees table.

SELECT \* FROM sales JOIN employees

ON sales.employee = employees.id;

Now that we are including multiple tables, we need to be explicit in our query with regard to which table or column we are referring to throughout the query. We can still specify columns in our **SELECT** and have conditions in our **WHERE**, we just need to clearly state which column or table we are talking about every step of the way. This is where the dot notation is so helpful. Let’s get the employee’s last name from the employees table, and the item from the sales table where the employee id is 114. Remember, with multiple tables, we now need to specify which tables/columns we are referencing in our query.

SELECT employees.last\_name, sales.item

FROM employees JOIN sales

ON sales.employee = employees.id

WHERE employees.id = 114;

Extending this technique, we can join more than two tables…

SELECT sales.date, employees.last\_name, stock\_items.item, stock\_items.inventory

FROM sales JOIN stock\_items

ON sales.item = stock\_items.id

JOIN employees

ON sales.employee = employees.id

WHERE sales.date = "2021-06-15";

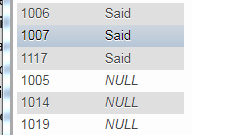
By default, the **JOIN** statement will only return rows where it finds a match in *both* tables. If a row in one table does not have a match in the other table, it will not become part of the result set. This is known as an “inner” join. In some situations, we will want an “outer” join that will return all of the rows from one table -- whether it finds a match or not in the other table. To create an outer join, we specify which table we want prioritized, the table on the **LEFT** side of the **JOIN** clause, or the table on the **RIGHT** side of the **JOIN** clause. Typically, if you want an outer join, it will be a **LEFT** join. **RIGHT** joins are usually reserved for subqueries. Below, we want to be sure to get all of the rows from the sales table - even if there isn’t a matching value in the employees table. If the **LEFT JOIN** doesn’t find a match in the right-side table, it will insert a null value in the result set.

SELECT sales.item, employees.last\_name

FROM sales LEFT JOIN employees

ON sales.employee = employees.id;

Below, is a screen capture of the last few rows of the result set, showing the null values created by the left join for the rows in the sales table that did not have a match in the employees table.



Queries Are More Easily Readable When Using Table Aliases

Within our **SELECT** statements, we created custom text for our column headings by creating an “alias” for that column with the **AS** keyword: SELECT category AS “Animal Group”…

We can do the same with our table names. This becomes especially important when you start joining multiple tables. It keeps your query much more readable - and less prone to typos! We create an alias for a table in the **FROM** clause by leaving a space after the table name, and then by providing the alias that we wish to use. Most commonly, a table is abbreviated to a single letter. For the three tables of our Pet Store database:

…FROM employees e

…FROM sales s

…FROM stock\_items i

Now, we can reference these tables by their one-letter alias in the rest of our query, following the same dot notation technique as we have been using. Note, although stock\_items starts with an S, I’m using S as the alias for sales, so I used I (for items) as the alias for stock\_items.

If we go back to our multi-table query from above, you can see how much tidier (and readable!) a query becomes by using aliases:

SELECT s.date, e.last\_name, i.item, i.inventory

FROM sales s JOIN stock\_items i

ON s.item = i.id

JOIN employees e

ON s.employee = e.id

WHERE s.date = "2021-06-15";

In some cases, (“…when we are looking for relationships between rows in a table”, according to Thomson and Welling), we want to compare values within the same table. This technique is referred to as a “self join”, and simply creates two copies of the *same* table by using *two* aliases for the *same* table. To find the items in the stock\_items table that have the same category as “Small cage” from the items column, we create an s1 and s2 alias for stock\_items, then proceed with our query:

SELECT s2.item, s2.category

FROM stock\_items s1, stock\_items s2

WHERE s1.item = "Small cage"

AND s1.category = s2.category;

References

Thomson, L., & Welling, L. (2003). Advanced queries, *MySQL tutorial* (pp. 95-101). Sams.

Thomson, L., & Welling, L. (2003). Database design crash course, *MySQL tutorial* (pp. 28-29). Sams.

The above reference is available through the Humber Library ([library.humber.ca](https://library.humber.ca/)).

https://www.w3schools.com/sql/sql\_join.asp