Section slides: http://webdev.slides.com/coltsteele/mysql-99-104

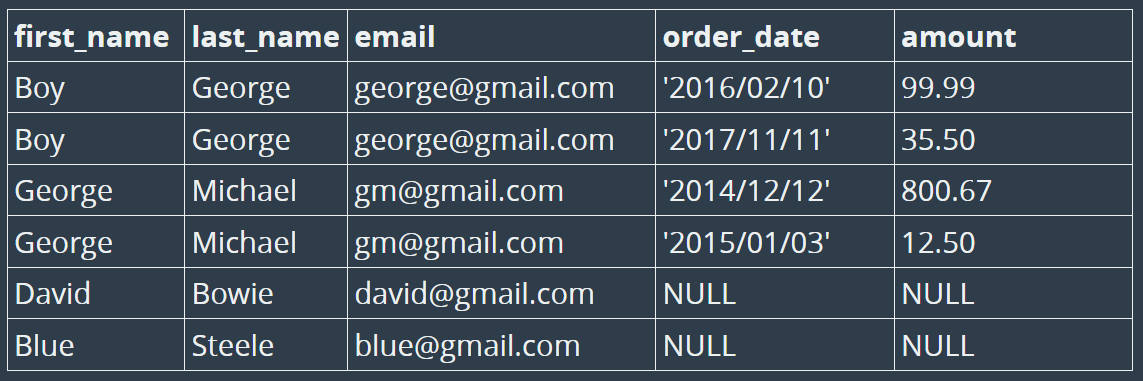
* In this section we will see how to relate tables to one another
* Prior to now, we were working with relatively simple data in only a single table, and those tables had only a few columns
* But in the real world, we can very complex data that is housed across multiple tables, and this inter-tabular data tends to be interrelated and dependent on each other
* Storing important information for a typical database may require half a dozen or more tables
* SQL Joins: <https://www.rdnaidu.com/2019/10/types-of-joins-in-sql.html>

# Types of Data Relationships

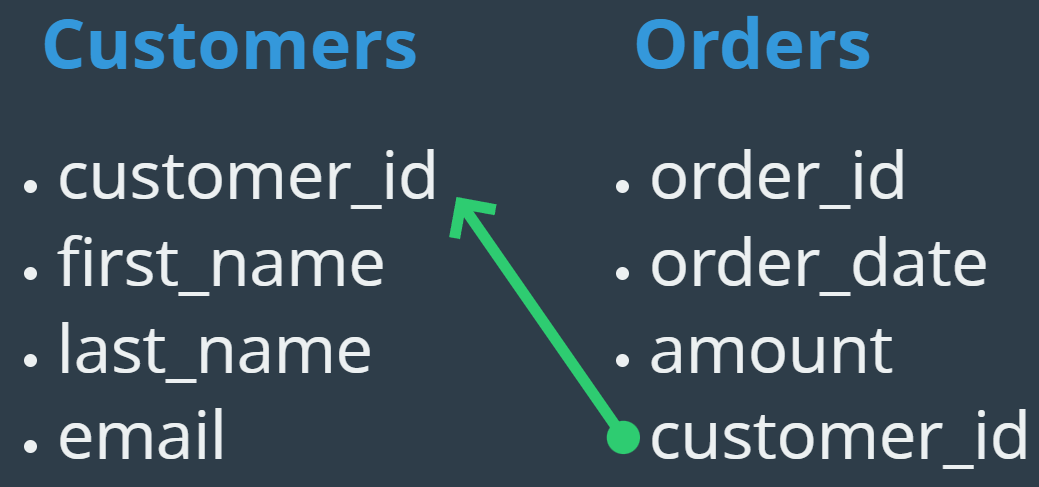
* In SQL, there are three broad categories of relationships
  + One to One: when each entry in one table is related to one and only one entry in another table
    - Ex: Customer ID Table to Customer Details table
  + One to Many: when one entry in one table is related to zero, one, or multiple entries in another table
    - Ex: A table of reviews for books to a table of books
    - The table of reviews can have multiple reviews for one book, but all of those reviews belong only to that one book
  + Many to Many: when many entries in one table are related to zero, one, or multiple entries in another table
    - Ex: In a database of books and authors, a given book can have multiple authors (books table), and each of those authors can be authoring multiple books independently (authors table)
  + More examples of relationships: <https://condor.depaul.edu/gandrus/240IT/accesspages/relationships.htm#:~:text=There%20are%20three%20types%20of,to%20the%20data%20and%20tables>.

# The Basics of One To Many

* 1:Many is perhaps the most common type of relationship in relational databases
* Consider an example situation where we have two tables: one of customers, and one of orders
  + Customers can have more than one order, but a given only has one customer associated with it
  + We want to store
    - A customer’s first and last names (VARCHAR)
    - A customer’s email (VARCHAR)
    - The date of the purchase (DATE)
    - The price of the order (DECIMAL)
* What would be a way to approach storing this data? One option is a single table where we capture every order. Notice how some customers have more than one order



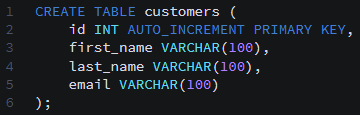
* + However, this is NOT a good idea.
    - There is duplication of some data, line names and emails
    - Some customers have not placed any orders, so those values are NULL. There is no reason to have data on orders for customers who have not placed any orders. Instead, it would be better to have customers tracked in a separate table
* The simplest approach to this issue is to create *two separate tables* for **Customers** and **Orders**, and they will be related to each other through **customer\_id**, which will be a column in BOTH tables
  + As a result of this reorganization, each customer in the *customers* table is unique and has a dedicated **customer\_id**
  + The *orders* table also has a **customer\_id** column – note that because a customer can place more than one order, a given customer\_id can appear more than once in this table. Similarly, any given order is associated with one and only one customer
  + Therefore, the relationship between these two tables is One (customers) to Many (Orders)
  + We also see that the customers who have not placed any orders are no long associated with NULL values as they were in the single table
  + Both tables have a PRIMARY KEY, which is a column that contains NO duplicate values and uniquely identifies that row of data
    - For the *customers* table, the PRIMARY KEY is **customer\_id**
    - For the *orders* table, the PRIMARY KEY is **order\_id**
  + Here we see our first **FOREIGN KEY**. A foreign key is a key that references the PRIMARY KEY of another table.
    - For our *orders* table, the FOREIGN KEY is **customer\_id**, as it refers to the customer\_id value in the *customers* table
    - *customers* does NOT have a FOREIGN KEY, as it does not have a column that references a PRIMARY KEY of the *orders* table
  + Foreign keys are important for database integrity. For example, we would not be able to add data to the *orders* table with a customer\_id that does not exist in the *customers* table



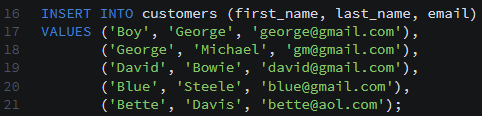


# Working with Foreign Keys

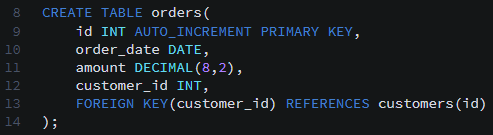
* For this section, we’ll be creating our *customers* and *orders* tables to demonstrate working with foreign keys. We’ll do this within a new database called **customers\_and\_orders**
* For *customers*, we’ll have the columns customer\_id, first\_name\_last\_name, and email. customer\_id will be an INT with auto-increment as well as our primary key, while the remainder will be VARCHAR
  + We will be referencing customer\_id as a foreign key from another table, and thus it must be unique

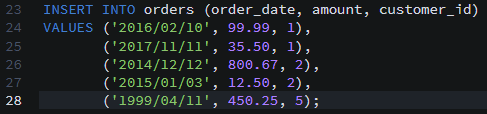


* + Inserting our data



* For *orders*, we’ll have id, order\_date, amount, and customer\_id. id will be an INTO and primary key, order\_date will be a DATE, and amount will be DECIMAL, and customer\_id will be INT and well as a FOREIGN KEY.
  + FOREIGN KEY is a new concept for us. The syntax involves using the REFERENCES keyword in our table creation, and ensure that the reference points to the customer\_id column of the *customers* table (see syntax below)
  + Why do we need to use FOREIGN KEY? The reason is that customer\_id is how we will be linking *customers* and *orders* together, and thus for the purposes of database integrity, any customer\_id that appears in *orders* must also exist in *customers*.
    - Important note: The variable names of the FOREIGN KEY and the PRIMARY KEY that it references do not need to be the same. In our example, customer\_id is simply “id” in the *customers* table but is “customer\_id” in the *orders* table
    - This is conventional when using a foreign key





* To test our database integrity, let’s try to insert an order for a customer that does not exist
  + The insertion will fail because we have no customer whose ID is 98



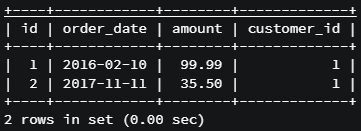


* We now have these tables associated with each other. In the next table we’ll explore JOINS, where we can do things like get the name of the customer based on orders they placed.

# Cross Join

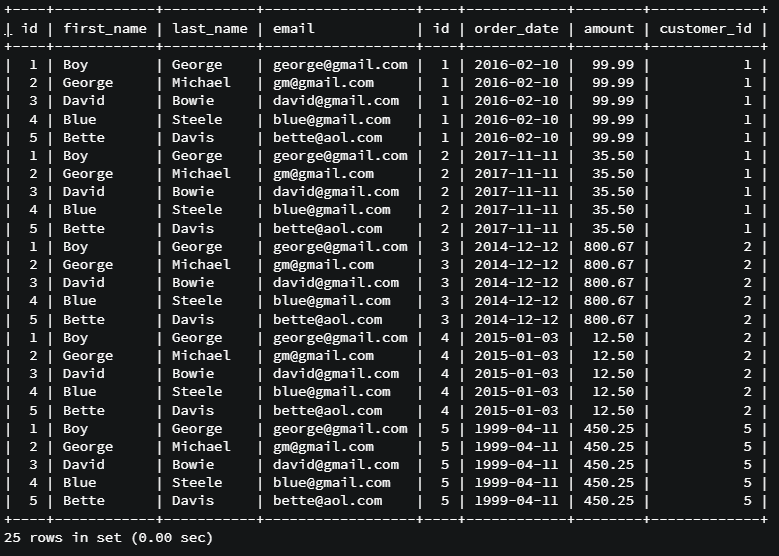
* Now that we have our two tables, let’s learn to do some queries where we’ll want to pull data from both tables!
* Suppose we want to *find the orders that were placed by Boy George*
  + One way to approach this is a two-step process, where we first search the *customers* table for the *customer\_id* that corresponds to Boy George. Then once we know that ID, we search the *orders* table for orders associated with Boy George’s customer ID.
  + We can also do it all at once with a subquery:





* + But this is a bit wordy, and it’s limited in its functionality. For example, what if we wanted to see not only information from the *orders* table, but also information from the *customers* table, like the name of the customer. That is, we’d like to see a **synthesis** of information from **both** tables
* We can do just that with **joins**. Joining allows us to conjoin two tables is different configurations and stick them together in a meaningful way.
* We’ll start with a **cross join**, also known as Cartesian join, which is quite useless but good to know.
  + The cross join takes every single item from the first table (customers) and conjoining it with every order (sort of like multiplying)
  + This is rarely an informative or useful process





* A far more useful procedure would be to perform a join where there is an actual overlap of information between the tables, say, orders that were ACTUALLY placed by Boy George. More on that in the next section
* Code summary:

**-- Finding Orders Placed By George: 2 Step Process**

1. SELECT id FROM customers WHERE last\_name='George';
2. SELECT \* FROM orders WHERE customer\_id = 1;

**-- Finding Orders Placed By George: Using a subquery**

1. SELECT \* FROM orders WHERE customer\_id =
2. (
3. SELECT id FROM customers
4. WHERE last\_name='George'
5. );

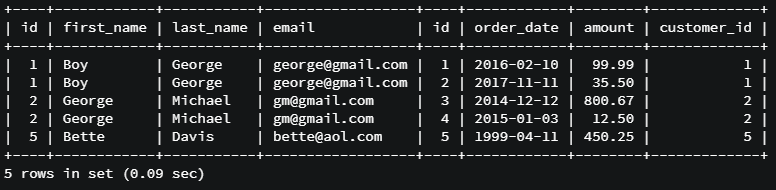
**-- Cross Join Craziness**

SELECT \* FROM customers, orders;

# Inner Join

* We’d like to see only orders where the “id” from the *customer* table matches the “customer\_id” from the *orders* table. This can be accomplished by using a WHERE statement. However, since both tables have a column called “id”, we need to specify which column needs to match the “customer\_id”
  + The workaround is very simple: **we preappend the name of the table that the column belongs to**





* Very nice. But notice how there’s some repetition here – customers.id matches orders.customer\_id, which is exactly what we wanted, but it does not necessarily need to be shown twice. Thankfully, we can clean this up by making or selection more targeted (i.e. instead of using \*)

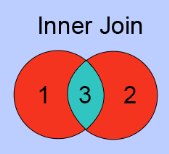




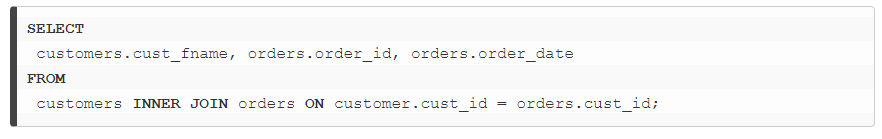
* + We could also have dropped the pre-appendices in the SELECT statement because there is no ambiguity between the column names in the two tables. If the tables did use the same column names, the pre-appendices would be needed. In this case, the statement below is equivalent to the one above.



* + This is known as an **implicit inner join**, that is, an inner join that is not explicitly declared yet functions exactly like an inner join. Only the data that satisfies the WHERE statement is included in the selection

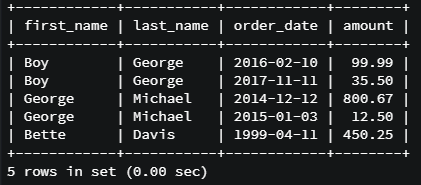


* There is also an **explicit inner join** which uses the keyword JOIN. Here’s how it’s done
  + Syntax:



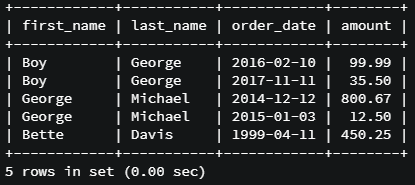
* + Example:





* Does order matter when making these joins? Yes and no.
  + The data will be exactly the same, but the way in which it is presented to you might differ, depending on how you do your selection





* Explicit joins are always better than implicit joins because it is much more clear what you are doing
* Inner join animation: <https://dataschool.com/how-to-teach-people-sql/inner-join-animated/>
* Code summary

**-- IMPLICIT INNER JOIN**

SELECT \* FROM customers, orders

WHERE customers.id = orders.customer\_id;

**IMPLICIT INNER JOIN**

SELECT first\_name, last\_name, order\_date, amount

FROM customers, orders

WHERE customers.id = orders.customer\_id;

**-- EXPLICIT INNER JOINS**

SELECT \* FROM customers

JOIN orders

ON customers.id = orders.customer\_id;

SELECT first\_name, last\_name, order\_date, amount

FROM customers

JOIN orders

ON customers.id = orders.customer\_id;

SELECT \*

FROM orders

JOIN customers

ON customers.id = orders.customer\_id;

**ARBITRARY JOIN - meaningless, but still possible**

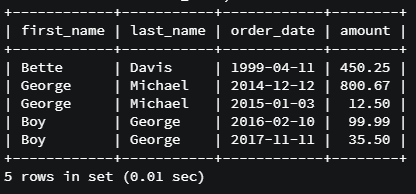
SELECT \* FROM customers

JOIN orders ON customers.id = orders.id;

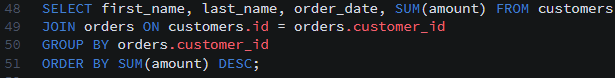
# Left Join

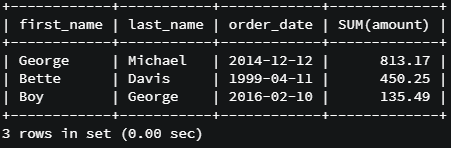
* Before we continue, it is worth mentioning that we can do anything we want with our joined table, including our previous manipulations like ORDER BY:



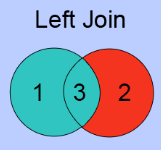


* + Or GROUP BY. Suppose we want to *group all orders by the customer who bought them and then determine who spent the most by doing an aggregate function, then order the customers by total amount spent on orders*
    - We have several options on how to do our customer grouping, but the best approach is to group by something that we know is unique – in this case, customer\_id (multiple people may have the same first and last names, for instance)



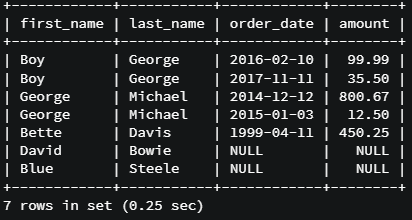


* The **LEFT JOIN** retrieves all records in the table that is on the (literally) LEFT side of the join condition AND…any columns that match from the table on the RIGHT side of the condition. So a so customer with no orders will be returned from the customer table, but NULL values are returned for the columns in the orders table for the rows that do not match the JOIN condition.

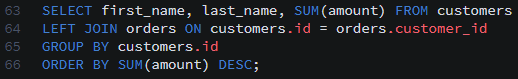


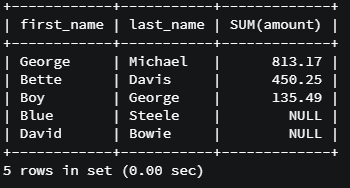
* Let’s now try a LEFT JOIN from our previous selection. The syntax is very similar, except that we specify that we want a LEFT JOIN instead of an INNER JOIN



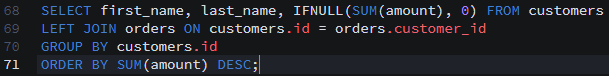


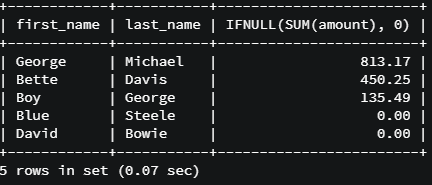
* + Notice the differences here – data from the LEFT table (*customers*) is included in the final selectionregardless of whether the ON condition is met. If there is no corresponding data from the *orders* table, those values are populated as NULL. In this case, David Bowie and Blue Steele did not have any orders in the *orders* table
    - Has this been an INNER JOIN, David Bowie and Blue Steele would not show up
* So why would you do a LEFT JOIN versus an inner join? Why would you want this excess information? It depends on what you’re doing – perhaps you want to tabulate the high spenders but also include those who have spent nothing. Let’s re-do our GROUP BY amount from above, but this time using a LEFT JOIN.
  + For this approach, we’ll GROUP BY customers.id to ensure that all customers get accounted for. Previously we used orders.customer\_id, which would exclude anyone who didn’t place any orders





* + But those NULLs are disgusting. What if we want to replace cases where no orders were placed by the customer with 0 instead of NULL? We can do this with the **IFNULL()** argument, which returns an expression if the value is NOT NULL, but returns the value if it is NULL
    - <https://www.w3schools.com/sql/func_mysql_ifnull.asp>





* LEFT and RIGHT JOIN animations: <https://dataschool.com/how-to-teach-people-sql/left-right-join-animated/>
* Code summary

-- Getting Fancier (Inner Joins Still)

SELECT first\_name, last\_name, order\_date, amount

FROM customers

JOIN orders

ON customers.id = orders.customer\_id

ORDER BY order\_date;

SELECT

first\_name,

last\_name,

SUM(amount) AS total\_spent

FROM customers

JOIN orders

ON customers.id = orders.customer\_id

GROUP BY orders.customer\_id

ORDER BY total\_spent DESC;

**LEFT JOINS**

SELECT \* FROM customers

LEFT JOIN orders

ON customers.id = orders.customer\_id;

SELECT first\_name, last\_name, order\_date, amount

FROM customers

LEFT JOIN orders

ON customers.id = orders.customer\_id;

SELECT

first\_name,

last\_name,

IFNULL(SUM(amount), 0) AS total\_spent

FROM customers

LEFT JOIN orders

ON customers.id = orders.customer\_id

GROUP BY customers.id

ORDER BY total\_spent;