Lesson 8

* In relational databases, data between different tables can be related using primary and foreign keys.
  + A primary key is the unique identifier for each row in a table. This is usually called ID or table name ID. In our employees table, it is the emp\_no.
  + The data from a specific row in the employees table can then be referenced in another table by using a foreign key, which means the other table has a column that holds values from the employees table’s emp\_no, or primary key.
  + **Use excel to demonstrate** For example, if we have a customers table, and each customer can have multiple phone numbers, it wouldn’t make sense to have a ton of columns named phone\_number\_1 …2 …3 …4 and so on, because each customer may not have that many numbers. Instead, we would create a phone\_numbers table that had a foreign key pointing to our customers table’s id, or primary key. That way we know, through the relationship formed by the primary and foreign key, which customer each phone number belongs to.
  + if you run `desc employees;` you will see that emp\_no is the primary key.
  + Run `show tables;` to see what other tables exist.
  + If you run `desc titles` you can see that the table has a column called emp\_no. This column references the emp\_no in our employees table. This lets us see the titles that each employee has had.
    - This is known as a one-to-many relationship. One employee can have many titles.
  + If you run `desc dept\_emp` you will see a emp\_no and a dept\_no. We already know what the emp\_no is, it references the employees table. Dept\_no references the departments table. The dept\_emp table is called a Join Table and allows us to create many-to-many relationships. One employee can be in many departments, and one department can have many employees. You need three tables (the two main tables and a join table) to create a many-to-many relationship.
    - The join table holds the relationships between the two table’s primary keys, thus tying an employee to a department on each row.
  + In summary, we use foreign keys to reference a tables primary key in a column of another table to create a relationship between a row in one table and a row in another.
* Now that we know how relationships are formed in tables, how do we use those relationships to query data across tables? To do this, we use something called a Join.
  + //Discuss types of joins by scrolling to bottom of <https://www.w3schools.com/sql/sql_join.asp>
  + MySQL supports inner join, left outer join, and right outer join
  + Select employees.\*, titles.\* from employees inner join titles on titles.emp\_no = employees.emp\_no;
    - There are a lot of columns there, we may not need them all. Let’s look at just the first and last name, emp\_no, and title.
  + Select employees.emp\_no, employees.first\_name, employees.last\_name, titles.title from employees inner join titles on titles.emp\_no = employees.emp\_no;
    - You can see how this could become verbose pretty quickly. We can make it less verbose by using an alias for the table name. An alias will allow us to shorten the table name in our query to avoid writing the full table name over and over.
  + Select e.emp\_no, e.first\_name, e.last\_name, t.title from employees e inner join titles t on t.emp\_no = e.emp\_no;
    - Notice how much shorter that is, a lot less verbose.
    - You define the alias right after the table names
  + You can also join multiple tables together as long as you have a reference, or key, from one table to the next. This allows you to combine related data across multiple tables.
  + Let’s look at using a couple joins to display data in a many to many relationship between employees and departments.
    - select d.dept\_name, e.first\_name, e.last\_name from departments d inner join dept\_emp de
    - -> on de.dept\_no = d.dept\_no
    - -> inner join employees e
    - -> on e.emp\_no = de.emp\_no;
  + If you get errors saying a table name is not recognized. Make sure your aliases are properly defined and that you are using the aliases instead of the full table name throughout the query.
  + You can use joins in all of your queries and you can add all the other clauses you’ve learned with them as well such as WHERE and ORDER BY.
* We can also perform functions in MySQL to change the way we look at data and help organize it to see specific patterns.
  + Count, min, max, avg, sum
  + Select count(\*) from salaries;
  + Select min(salary) from salaries;
  + Select max(salary) from salaries; avg and sum too
  + Using aliases for column names with the AS statement is useful here: select max(salary) as ‘Max Salary’ from salaries;
  + Select concat(first\_name, “ “, last\_name) as “Full Name” from employees;
  + There are a lot of functions out there. Take some time to research and see how they are used.
  + We can also use some of these functions as aggregate functions, meaning we can group data together based on the results of these functions.
    - Count, min, max, avg, sum are examples of functions that can be used to aggregate data. For example, you could aggregate the count of each salary to see how many employees are in each salary. To do this we have to use a new statement GROUP BY. This allows us to aggregate data and group it by a specific column. In this example, we will aggregate the count of salary grouped by salary. This will show two result columns, the count, or number of salaries, next to the actual salary itself.
    - select count(salary) as "Number of Salaries", salary from salaries group by salary;
    - let’s get a little more advanced and write a query to show the average salary per department.
      * To start, let’s look at the salaries and departments tables and see if we can find a relationship
      * Running `desc` for each table shows that there is not a direct relationship between the two tables. Neither refers to the others primary key. But we see that salaries refers to emp\_no (which is the primary key for employees) and we know that departments has a relationship to employees through the join table dept\_emps. We can follow the keys and tie all these tables together.
        + Departments joins to dept\_emps on dept\_no
        + Dept\_emps can then join to salaries on emp\_no
      * select d.dept\_name as "Department", avg(s.salary) as "Average Salary" from departments d
      * -> inner join dept\_emp de on de.dept\_no = d.dept\_no
      * -> inner join salaries s on s.emp\_no = de.emp\_no
      * -> group by d.dept\_name;
      * Now say you only want to see departments with an average salary over 60000. Normally you would think to use a WHERE clause, but aggregation doesn’t work with a WHERE clause, instead we use HAVING, which works just like a where clause, but for aggregated data.
        + Note that you can still have a WHERE clause in a query that uses aggregation, but not to add a condition as to which aggregates to show, rather you can still use it in the initial part of the query to determine which data to aggregate in the first place.
        + SELECT column\_name(s)  
          FROM table\_name  
          WHERE condition  
          GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);
      * Let’s show only departments with an average salary of 60000 or higher.
        + select d.dept\_name as "Department", avg(s.salary) as "Average Salary" from departments d
        + -> inner join dept\_emp de on de.dept\_no = d.dept\_no
        + -> inner join salaries s on s.emp\_no = de.emp\_no
        + -> group by d.dept\_name
        + -> having avg(s.salary) >= 60000;
      * You can explore the different aggregate functions and see what the results are.