Introduction to SQL

Relational Databases (1/4)

- A relational database defines database relationships in the form of tables. The tables are related to each other - based on data common to each.

Relational Databases (2/4)

Customers Table

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Source: w3schools – MySQL RDBMS

Relational Databases (3/4)

Shippers Table

ShipperID	ShipperName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

Relational Databases (4/4)

Orders Table

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10278	5	8	1996-08-12	2
10280	5	2	1996-08-14	1
10308	2	7	1996-09-18	3
10355	4	6	1996-11-15	1
10365	3	3	1996-11-27	2
10383	4	8	1996-12-16	3
10384	5	3	1996-12-16	3

Tables

- Every relational database stores information in tables. You can have many tables in one database and each of your tables will hold data which refers to similar objects. Each table has a name so you can find out what kind of information is stored there.
- For example, the database of your university would include a table named student with all data regarding students, another table subject with information on the subjects at your university, etc.

Columns and rows

- Tables in databases look exactly the way you would imagine a normal table – they have **columns** and **rows**.
- Columns in every table have their **names** and they identify the kind of information is stored in them.
- Each row stores information about one object. In the student table below, you can see that the column names reflect the kind of data contained in them, so in the column name there are names of students, in the column graduation_year there is the year of their graduation, etc.

Each row corresponds to exactly one student.

id	name	year_born	graduation_year
1	Mohammed Salah	1992	2015
2	Lionel Messi	1991	2013
3	Ebrahim Sombol	2000	2024
4	Linus Torvalds	1969	1996
5	Marwan Moussa	1995	2019

SQL

- So, how do we get in touch with our database? We use the so-called **Structured Query Language**. Of course, no one uses the full name, we just call it SQL for short.
- All relational databases understand SQL, but each of them has a slightly different dialect, so to speak.
- We will learn the basics of the standard SQL which will be understood by every relational database. Thanks to SQL, you'll be able to make queries in each database environment.

Queries

- Queries are questions that we ask to find out some information about the data stored in the database.
- Databases can do amazing things – they don't only return the data you ask for, they can actually do advanced calculations on the tables. You'll see for yourself!
- By the end, you should be able to write fairly complex queries. Ready?

Let’s get all data

- You can see all data in the STUDENT table with this query:

```
SELECT *
FROM STUDENT;
```

- SELECT tells your database that you want to select data. FROM user tells the database to select data from the table STUDENT.
- the asterisk (*) tells the database that you want to see all columns in this table.
- Remember to add semicolon in the end. It’s a good practice.

Let’s get one column

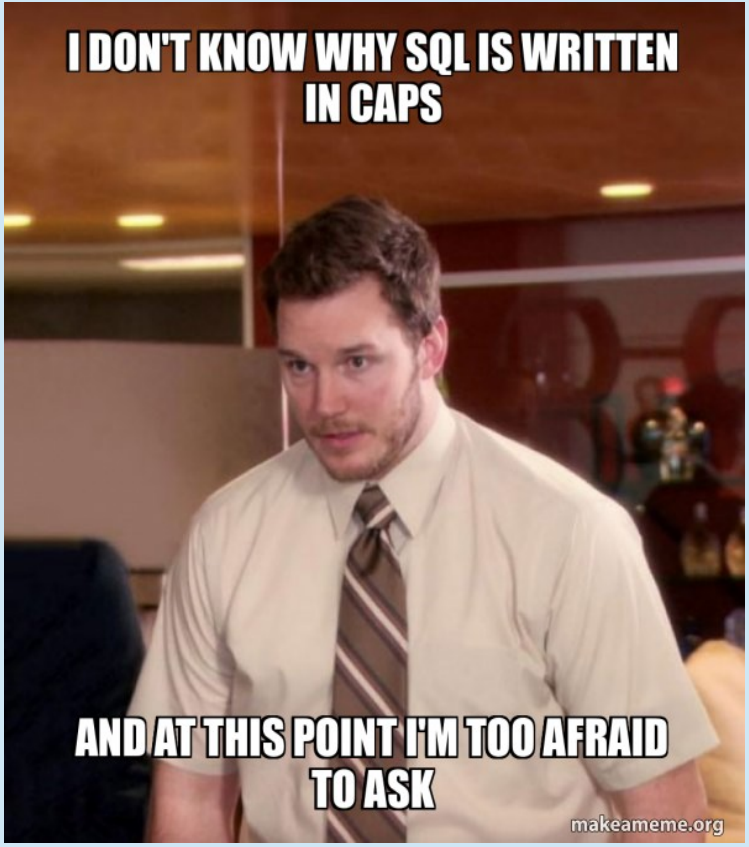
- What if you don't want to select all columns from a table? No problem. Just type the column name instead of the asterisk. If you want to get names of all users, type:

```
SELECT name
FROM STUDENT;
```

Let's get many columns

- If you want to get **a couple of columns**, give the names of the desired columns before FROM. So to get names and ages of all users, type:

```
SELECT
      name,
      year_born
FROM student;
```



Select only a few rows

- What if we just wanted the users who are born in 2006, in our small table we can select all* and get what we need.
- But what if the table consists of thousands of rows??

```
SELECT *
FROM STUDENT
WHERE year_born = 2006;
```

- WHERE is like condition

Conditional operators

- Besides the equlity sign (=), There are also some older conditional operators which you can use.

```
SELECT *
FROM STUDENT
WHERE year_born <= 2006;
```

- WHERE is like condition

Conditional operators

- Other conditionl operators:-
  - (<) → less than
  - (>) → greater than
  - (<=) → less than or equal to
  - (>=) → greater than or equal to
- Let’s play a bit with our table.

The not equal sign (!=)

- here is one more very important conditional operator, the inequality sign (!= or sometimes <>). Look at the example:
-

```
SELECT *
FROM STUDENT
WHERE year_born <> 2006;
```

Conditional operators and selecting columns

- Let's combine what we know about conditional operators with selecting specific columns.
-

```
SELECT
      id,
      year_born
FROM STUDENT
WHERE year_born >= 2006;
```

Logical operators – OR

- What about situations when we want to be really picky and join a few conditions?
-

```
SELECT *
FROM STUDENT
WHERE year_born < 2006
      OR salary > 400;
```

How OR Works??

```
True  + True  = True
True  + False = True
False + True  = True
False + False = False
```

Logical operators – AND

- OR isn't the only logical operator we have, there is also the AND logical operator

```
SELECT *
FROM STUDENT
WHERE year_born <= 2006
      AND salary > 200;
```

How AND Works??

```
True  + True  = True
True  + False = False
False + True  = False
False + False = False
```

The BETWEEN operator

- What if we want to get the people who are born between 2000 and 2006? Well, We have two methods

```
SELECT *
FROM STUDENT
WHERE year_born <= 2006
      AND year_born >= 2000;
```

```
SELECT *
FROM STUDENT
WHERE year_born
      BETWEEN 2006 AND 2000;
```

- IMP) Not all databases produce the same output with BETWEEN, some include 2006 and 2000, some don't

Logical operators – NOT

- There is one more logical operator worth mentioning: NOT.
- Whatever is stated after NOT will be negated:

```
SELECT *
FROM STUDENT
WHERE year_born
      NOT BETWEEN 2000 AND 2006;
```

Join even more conditions

- We can include even more conditions by using **parentheses**, according to our needs.
-

```
SELECT *
FROM STUDENT
WHERE (year_born <= 2000 AND year_born >= 1990)
      OR salary > 10000;
```

Use text

- Until now, we only worked with numbers in our WHERE clauses. Is it possible to use letters instead?

```
SELECT *
FROM STUDENT
WHERE name = ‘Megahed’;
```

- Note: 1) Single quotes
  2) Case senitivity

The percentage sign (%)

- What if we don't know precisely what letters we're looking for? With text values, you can always use the LIKE operator instead of the equality sign.

```
SELECT *
FROM STUDENT
WHERE name LIKE 'M%';
```

The underscore sign (_)

- Sometimes we may not remember just one letter of a specific name. Imagine we want to find a some whose name is... Islam? Eslam?

```
SELECT *
FROM STUDENT
WHERE name LIKE ‘_slam’;
```

Looking for NOT NULL values

- In every row, there can be NULL values, i.e. fields with unknown, missing values.
- To check whether a column has a value, we use a special instruction IS NOT NULL.

```
SELECT *
FROM STUDENT
WHERE salary IS NOT NULL;
```

Looking for NULL values

- NULL is a special value. You can't use the equal sign to check whether something is NULL. It simply won't work. The opposite of IS NOT NULL is IS NULL.

```
SELECT *
FROM STUDENT
WHERE SALARY IS NULL;
```

Comparisons with NULL

- If you set a condition, say salary < 3000, the rows where salary is NULL will always be excluded from the results.
- In no way does NULL equal zero. What's more, the expression NULL = NULL is never true in SQL!

Gus Fring 🤨

Basic mathematical operators

- Problem: **simple mathematics**. Can you add or multiply numbers in SQL?

```
SELECT *
FROM STUDENT
WHERE (salary * 12) > 50000;
```

- + for addition, - for substraction, / for division, * for multiplication

PRACTICE TIMEEE!!

- Select all columns of students:-
  - Who are born between 1990 and 2000
  - Who are not from Syria
  - Whoose names start with ‘A’ or ‘M’
  - Who have a salary

SOLUTION

```
SELECT *
FROM STUDENT
WHERE (born_year BETWEEN 1990 AND 2000)
      AND (country IS NOT ‘Syria’)
      AND (name LIKE ‘A%’ OR ‘M%’)
      AND (salary IS NOT NULL);
```

What we have learned so far?

- We learned how to get data from a single table. We have also learned how to filter rows to only get those which you really need.
- Now, What's next?
  - Single tables might seem handy at first, but in big databases we always use multiple tables. This also means that we often want to get data from more than one table at a time.

Let's work in a bigger database.

- We are gonna use another database called HR.
- Before we can do anything, we must have admin privileges.
- Let's first unlock the database. How can we do that??
  - ALTER USER account ACCOUNT UNLOCK;
- We also need to set a password.
  - ALTER USER user_name IDENTIFIED BY new_password;
- Now, we are ready to use the HR database

Refer to Using SQL*Plus to Unlock Accounts and Reset Passwords for more information.

Multiple tables in the clause FROM

- There are quite a few ways of getting information from multiple tables at the same time.

```
SELECT *
FROM EMPLOYEES, JOBS;
```

- Before running the command, can you predict how many rows are we gonna get??

Join tables on a condition

- We get count(EMPLOYEES) * count(JOBS)
  - That’s 107 * 19 = 2033 row Ignoring the header row
- Why did this happen? SQL doesn't know what to do with the results from the two tables, so it gave you every possible connection. How can we change it?

```
SELECT *
FROM employees, jobs
WHERE employees.job_id = jobs.job_id;
```

The keyword JOIN

- Excellent! Now the result looks much better, right?

- Joining two tables is such a popular and frequent operation that SQL provides a special word for it: JOIN. There are many versions of JOIN out there.

Join tables using JOIN

```
SELECT *
FROM EMPLOYEES
JOIN JOBS
    ON EMPLOYEES.JOB_ID = JOBS.JOB_ID;
```

- We want to join the tables EMPLYEES and JOBS, so we use the keyword JOIN between their names.
- SQL must also know how to join the tables, so there is another keyword ON. After it, we set our condition: join only those rows where the job id in EMPLYEES is the same as the job id in JOBS.

Display specific columns

- Now, let's say we only need a few columns in our result. We just need the name of the empolyee and his job title.
-

```
SELECT
    EMPLOYEES.FIRST_NAME,
    JOBS.JOB_TITLE
FROM EMPLOYEES
JOIN JOBS
    ON EMPLOYEES.JOB_ID = JOBS.JOB_ID;
```

Rename columns with AS

```
SELECT FIRST_NAME AS NAME
FROM EMPLOYEES;
```

- The new name is just an alias, which means it's temporary and doesn't change the actual column name in the database. It only influences the way the column is shown in the result of the specific query. This technique is often used when there are a few columns with the same name coming from different tables. Normally, when SQL displays columns in the result, there is no information about the table that a specific column is part of.

Filter the joined tables

- Now that we know how to work with columns, let's find out how to filter the results even further:

```
SELECT
    EMPLOYEES.FIRST_NAME,
    JOBS.JOB_TITLE
FROM JOBS
JOIN EMPLOYEES
    ON EMPLOYEES.JOB_ID = JOBS.JOB_ID
WHERE EMPLOYEES.SALARY > 15000;
```

Let's practice

- Get the full name and hire date from EMPLOYEE table, and the job title from the JOBS table in such a way that an employee is shown together with its job title.
- Show the column JOB_TITLE as WHAT_WORK.
- Select only those employees whoose names starts with 'D' or the second letter of their name is 'a'

SOLUTION!!

```
SELECT
    EMPLOYEES.FIRST_NAME || ' ' || EMPLOYEES.LAST_NAME
    AS FULL_NAME,
    EMPLOYEES.HIRE_DATE,
    JOBS.JOB_TITLE as WHAT_WORK
FROM EMPLOYEES
JOIN JOBS
    ON EMPLOYEES.JOB_ID = JOBS.JOB_ID
WHERE EMPLOYEES.FIRST_NAME LIKE 'D%'
    OR EMPLOYEES.FIRST_NAME LIKE '_a%';
```

What we have learned so far?

- We have learned how to select data from single and multiple tables. We know how to filter rows and join columns from different tables. Now it's time to move on.
- What’s next?
  - We will learn how to compute **statistics**, **group** rows, and **filter** such groups. Such operations are extremely important for preparing **reports** and always come in handy in big tables.

Sort the rows – ORDER BY (1/2)

- Have you wondered how tables are **sorted** in the result of an SQL query? Well, the answer is simple – **by default, they are not sorted at all.**
- The sequence in which rows appear is arbitrary and every database can behave differently.
- You can even perform the same SQL instruction a few times and get a different order each time – unless you ask the database to sort the rows, of course.

Sort the rows – ORDER BY (2/2)

- If we want to order the employees by salary

```
SELECT FIRST_NAME, SALARY
FROM EMPLOYEES
ORDER BY SALARY;
```

- In what order the data will be sorted in?
  - (Ascending / Decreasing)??

ORDER BY with conditions

- We can filter rows and sort them at the same time.

```
SELECT FIRST_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 60
ORDER BY SALARY;
```

Ascending and descending orders

- We can reverse the order and make the greatest values appear first.

```
SELECT FIRST_NAME, SALARY
FROM EMPLOYEES
ORDER BY SALARY DESC;
```

- DESC → Descending Order
- ASC → Ascending Order

Sort by a few columns

- You can sort your results by **more than one column** and each of them can be sorted in a **different** order:

```
SELECT FIRST_NAME, DEPARTMENT_ID, SALARY
FROM EMPLOYEES
ORDER BY DEPARTMENT_ID ASC,
      SALARY DESC;
```

- It will first be sorted by DEPARTMENT_ID in the ascending order and then, for each DEPARTMENT_ID, the orders will be sorted by the SALARY in the descending order.

Duplicate results

- By default, the database returns every row which matches the given criteria. This is what we normally expect, of course, but there are cases when we might want to change this behavior.
- Imagine the following situation: we want to get the ids of all jobs from the EMPLOYEES table. We might use the following code:

```
SELECT JOB_ID
FROM EMPLOYEES;
```

- What's wrong with the code in this case?

Select distinctive values

- In the last slide we have seen that the job id get duplicated despite the fact that we only want to know the job IDs. Duplication was unnecessary. How do we avoid that??

```
SELECT DISTINCT JOB_ID
FROM EMPLOYEES;
```

- The DISTINCT keyword will help us **remove duplicates**.

Count the rows

- Our database can **compute statistics** for multiple rows. This operation is called **aggregation**.

```
SELECT COUNT(*)
FROM EMPLOYEES;
```

- COUNT(*) is a function. A function in SQL always has a name followed by parentheses. In the parentheses, you can put information which the function needs to work. For example, COUNT() calculates the number of rows specified in the parentheses.

Count the rows, ignore the NULLS

- Naturally, the asterisk (*) isn't the only option available in the function COUNT(). For example, we may ask the database to count the values in a specific column:

```
SELECT COUNT(COMMISSION_PCT)
FROM EMPLOYEES;
```

- What's the difference between COUNT(*) and COUNT(COMMISSION_PCT)??
  - The first counts all rows in the table, the ignores the rows that has COMMISSION_PCT set to NULL.

Find the min and max value

- The function MIN(SALARY) returns the **smallest value** in the column SALARY, While MAX(SALARY) returns the **biggest value** in the column.
-

```
SELECT MIN(SALARY)
FROM EMPLOYEES;
```

```
SELECT MAX(SALARY)
FROM EMPLOYEES;
```

Find the average value

- The function AVG() finds the average value of the specified column.
- Let’s get the average for employees who work as programmers.

```
SELECT AVG(SALARY)
FROM EMPLOYEES
WHERE JOB_ID = ‘IT_PROG’;
```

Find the sum

- To get the **total of values** in a specific column we use the function SUM().
- If we want to get the total of salaries given to the programmers in our company, we run the command
- ```
  SELECT SUM(SALARY)
  FROM EMPLOYEES
  WHERE JOB_ID = 'IT_PROG';
  ```
  ```

Group the rows and count them

- The new piece is GROUP BY followed by a column name JOB_ID. GROUP BY will group together all rows having the same value in the specified column.

```
SELECT JOB_ID, COUNT(*)
FROM EMPLOYEES
GROUP BY JOB_ID;
```

- All emplyees in the same JOB_ID will be grouped together in one row. The function COUNT(*) will then count all rows for the same JOB_ID.

Find the average value in groups

- To get the average of salaries in each JOB_ID, we run the command:

```
SELECT JOB_ID, AVG(SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID;
```

Group by a few columns

- Sometimes we want to group the rows by more than one column.

```
SELECT DEPARTMENT_ID, JOB_ID, SUM(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID, JOB_ID;
```

- Remember: in such queries each column in the SELECT part must either be **used later for grouping** or **it must be used with one of the functions**.

Filter groups

- Let’s practice on the LOCATIONS table.
- We want to get the number of available locations in each country. How can we do that??
- Can we also add a condition so we don’t get the number of locations in India or japan?
  - Psssst, the word is HAVING

SOLUTION

```
SELECT COUNTRY_ID, COUNT(LOCATION_ID)
FROM LOCATIONS
GROUP BY COUNTRY_ID
HAVING COUNTRY_ID <> 'JP' AND COUNTRY_ID != 'IN';
```

- Is this the same as this ?

```
SELECT COUNTRY_ID, COUNT(LOCATION_ID)
FROM LOCATIONS
WHERE COUNTRY_ID <> 'JP' AND COUNTRY_ID != 'IN';
GROUP BY COUNTRY_ID;
```

Order groups

- Groups can be sorted just like rows.
-

```
SELECT JOB_ID, SUM(SALARY) as TOTAL_SALARY_GROUP
FROM EMPLOYEES
GROUP BY JOB_ID
ORDER BY SUM(SALARY) DESC;
```

PRACTICE TIMEEE!!!

- Select the department name form the departments table, count how many employees in each department and display it as EMP_COUNT, get the total salaries in each department and display it as TOTAL_SALARIES.
- Only display those departments who has a total of salaries above 20000.
- Order the results according to the number of employees in each department in descending order.

Stuck? Here is the solution!

```
SELECT DEPARTMENT_NAME,
    COUNT(EMPLOYEE_ID) AS EMP_COUNT,
    SUM(SALARY) AS TOTAL_SALARIES
FROM EMPLOYEES, DEPARTMENTS
WHERE DEPARTMENTS.DEPARTMENT_ID = EMPLOYEES.DEPARTMENT_ID
GROUP BY DEPARTMENT_NAME
HAVING SUM(SALARY) > 20000
ORDER BY COUNT(EMPLOYEE_ID) DESC;
```

JOIN.. REMEMBER

- We are going back to the LEARN user.
- Join the two tables: student and room so that each student is shown together with the room they live in. Select all columns.

```
SELECT *
FROM STD
     JOIN ROOM
     ON STD.ROOM_ID = ROOM.ID;
```

JOIN.. REMEMBER

- You can select some columns in a JOIN query.

```
SELECT name, room_number
FROM std
JOIN room
    ON std.room_id = room.id;
```

- Instead of all columns (selected with the asterisk *) here we select two columns: name, room_number

INNER JOIN (1/4)

- JOIN is actually just one, of a few joining methods. It's the most common one so it's always applied **by default** when you write the keyword JOIN in your SQL statement. Technically speaking, though, its full name is INNER JOIN.
- This command is the same as in the previous slide

```
SELECT name, room_number
FROM std
INNER JOIN room
      ON std.room_id = room.id;
```

INNER JOIN (2/4)

- If you now compare the results of INNER JOIN with the content of the equipment table, you'll notice that not all pieces of equipment are present in the resulting table. For example, a lovely kettle with the id 11 is not there. Do you know why?
- INNER JOIN only shows those rows from the two tables where **there is a match between the columns**. In other words, you can only see those pieces of equipment which have a room assigned and vice versa. Equipment with no room is not shown in the result. Take a look at the table:

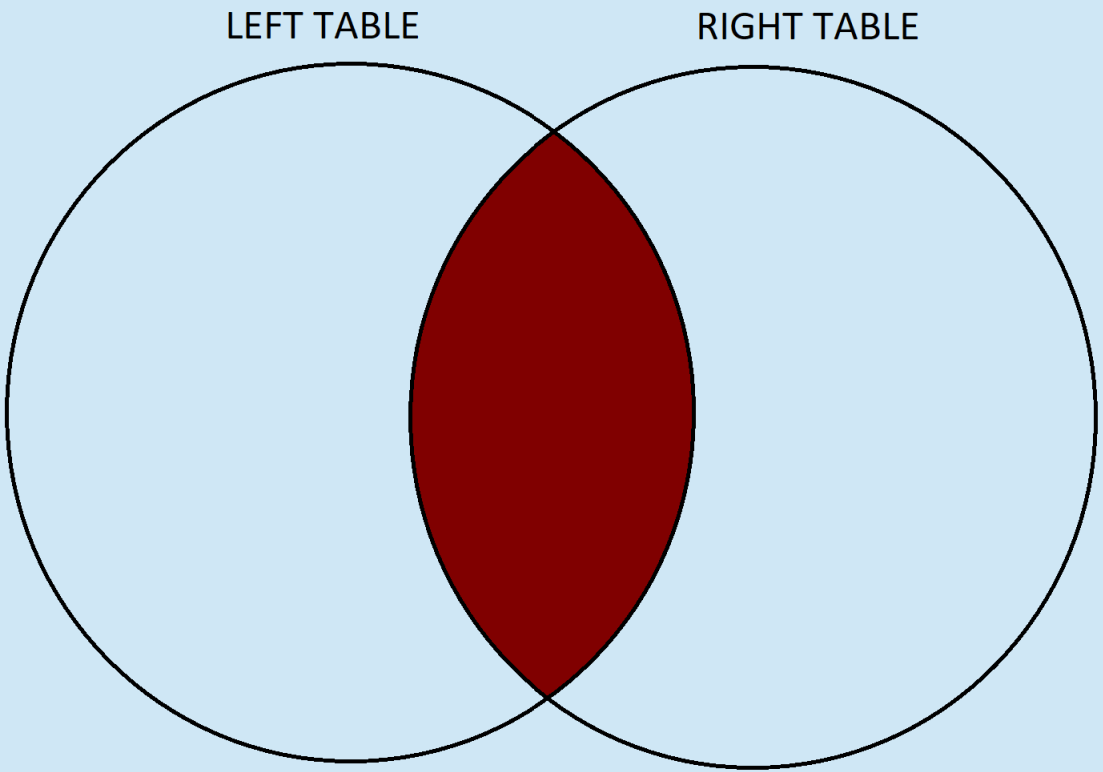
INNER JOIN (3/4)

- If we run the command

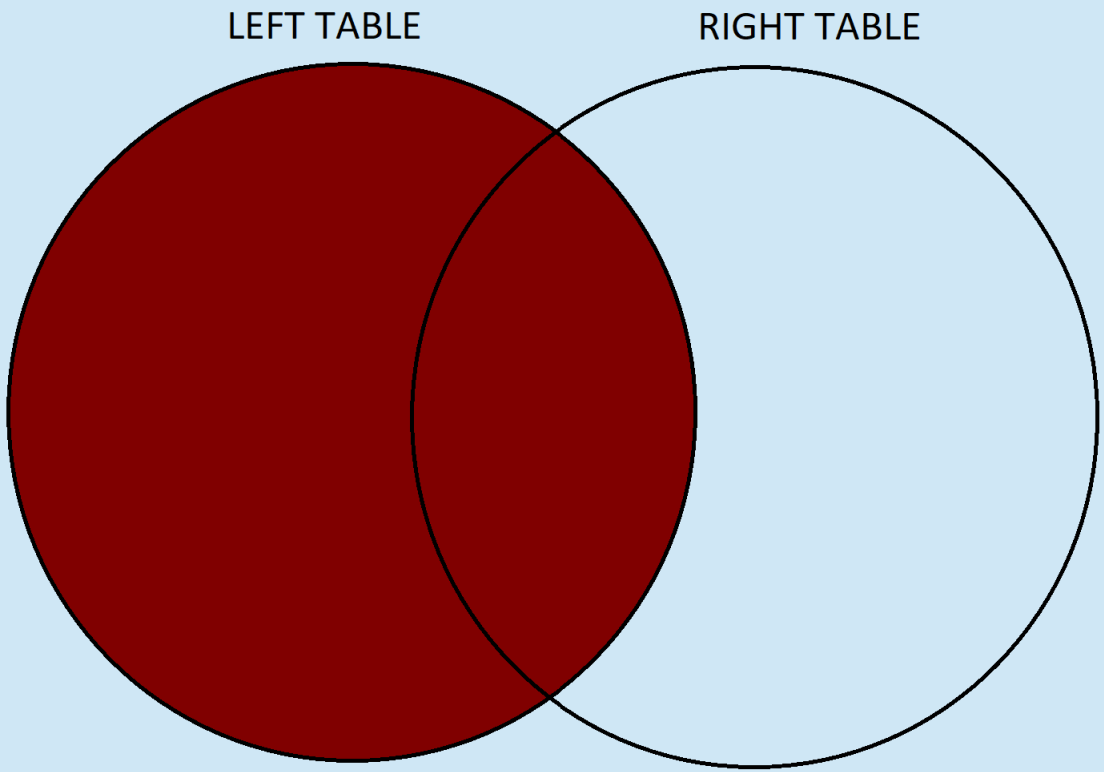
```
SELECT *
FROM equipment
INNER JOIN room
  ON equipment.room_id = room.id;
```

- Can you guess how many rows are we gonna get?

INNER JOIN (4/4)



LEFT JOIN (1/3)



LEFT JOIN (2/3)

- LEFT JOIN works in the following way: it returns all rows from the left table (the first table in the query) plus all matching rows from the right table (the second table in the query).

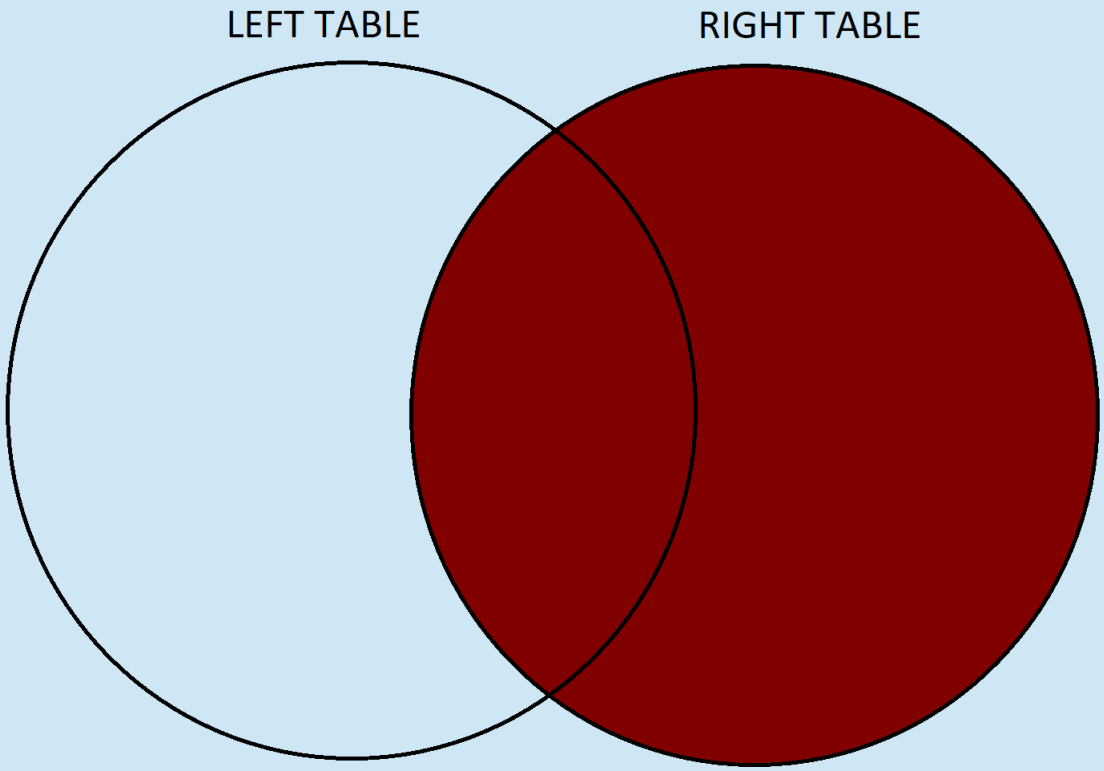
```
SELECT *
FROM std
LEFT JOIN room
      ON std.room_id = room.id;
```

LEFT JOIN (3/3)

- Select all pieces of equipment together with the room they are assigned to. Show each piece of equipment even if it isn't assigned to a room. Select all available columns.

```
SELECT *
FROM equipment
LEFT JOIN room
    ON equipment.room_id = room.id;
```

RIGHT JOIN (1/3)



RIGHT JOIN (2/3)

- The RIGHT JOIN works in the following way: it returns **all** rows from the **right table** (the second table in the query) plus all matching rows from the **left table** (the first table in the query).

- Practice: for each student show their data with the data of the room they live in. Show also rooms with no students assigned.

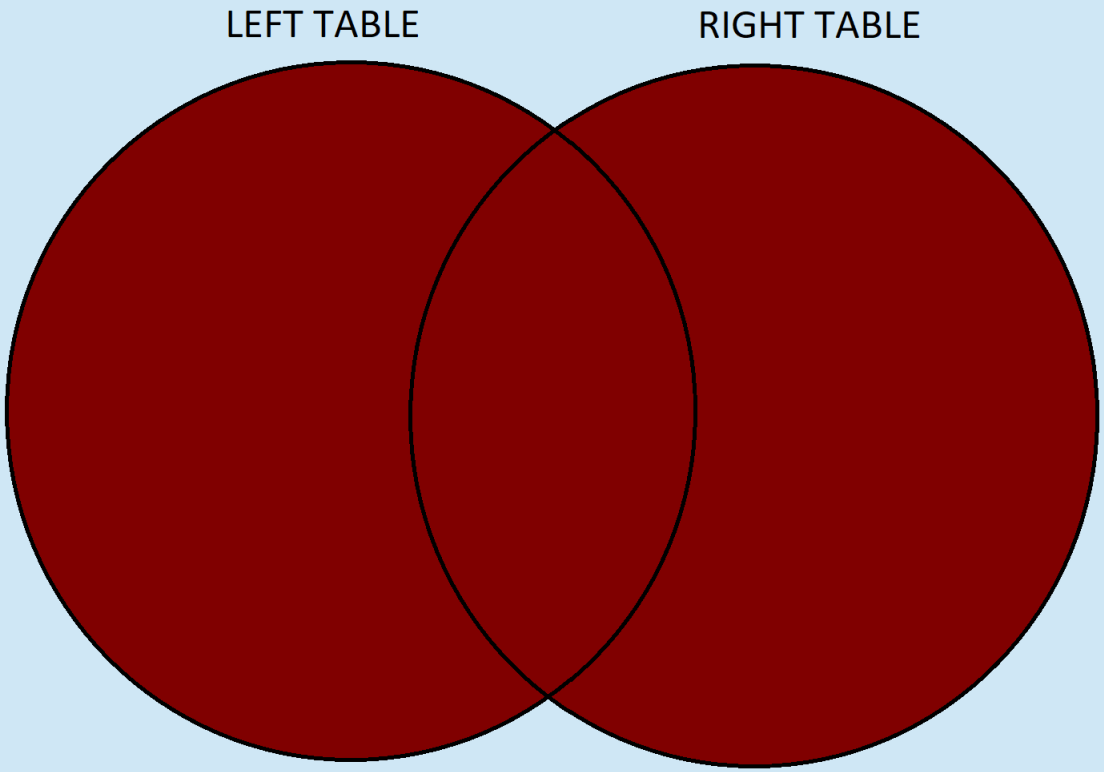
RIGHT JOIN (3/3)

- SOLUTION!!

```
SELECT *
FROM std
RIGHT JOIN room
      ON std.room_id = room.id;
```

- For each student show the room data the student is assigned to. Show also students who are not assigned to any room. Use a RIGHT JOIN.

FULL JOIN (1/2)



FULL JOIN (2/2)

- Another joining method is FULL JOIN. This type of JOIN **returns all rows from both tables** and combines the rows when there is a match. In other words, FULL JOIN is a union of LEFT JOIN and RIGHT JOIN.

The keyword OUTER

- Nothing new..
  - LEFT OUTER JOIN == LEFT JOIN
  - RIGHT OUTER JOIN == RIGHT JOIN
  - FULL OUTER JOIN == FULL JOIN
- Nothing new, what we were using in all of the past slides was just a shortcut.

NATURAL JOIN (1/3)

- There's one more joining method before you go. It's called NATURAL JOIN and it's slightly different from the other methods because it doesn't require the ON clause with the joining condition:

```
SELECT *
FROM std
NATURAL JOIN room;
```

NATURAL JOIN (2/3)

- NATURAL JOIN doesn't require column names because it always joins the two tables on the **columns with the same name**.
- In our example, students and rooms have been joined on the column ID, which doesn't really make much sense.

```
SELECT *
FROM std
NATURAL JOIN room;
```

==

```
SELECT *
FROM std
JOIN room
  ON std.id = room.id;
```

NATURAL JOIN (3/3)

- You can, however, construct your tables in such a way that NATURAL JOIN comes in handy. If you had the following tables:
  - car(car_id, brand, model)
  - owner(owner_id, name, car_id)
- Then it would make perfect sense to use NATURAL JOIN because it would join the two tables on the car_id column. You would then need fewer keyboard strokes to join two tables.

Table aliases

- What if I told you that, these two commands are the same!

```
SELECT
  person.id,
  person.name,
  person.year,
  car.id,
  car.name,
  car.year
FROM person
JOIN car
  ON person.id = car.owner_id;
```

```
SELECT
  p.id,
  p.name,
  p.year,
  c.id,
  c.name,
  c.year
FROM person AS p
JOIN car AS c
  ON p.id = c.owner_id;
```

*In Oracle, you need to use just the alias without the AS keyword.

Table aliases.. Practice!!

- Use INNER JOIN on the tables room and equipment so that all pieces of equipment are shown with their room data. Use table aliases r and e. Select the columns id and name from the table equipment, as well as room_number and beds from the table room.

```
SELECT e.id,
  e.name,
  r.room_number,
  r.beds
FROM room r
JOIN equipment e
  ON r.id = e.room_id
```

Aliases in self-joins

- We want to know who lives with the student Jack Pearson in the same room. Use self-joining to show all the columns for the student Jack Pearson together with all the columns for each student living with him in the same room.

```
SELECT *
FROM std s1
JOIN std s2
      ON s1.room_id = s2.room_id
WHERE s1.name = 'Jack Pearson'
AND s1.id != s2.id;
```

Joining more tables

- You can also use more than one join in your SQL instruction. Let's say we also want to show all the room information for the students paired with Jack Pearson. Unfortunately, data like room number or floor is not stored in the table student – we need yet another join with the table room:

Solution

```
SELECT *
FROM student s1
JOIN student s2
  ON s1.room_id = s2.room_id
JOIN room
  ON s2.room_id = room.id
WHERE s1.name = 'Jack Pearson'
      AND s1.name != s2.name;
```

Joining more tables.. Challenge

- For each room with 2 beds where there actually are 2 students, we want to show one row which contains the following columns:
  - the name of the first student.
  - the name of the second student.
  - the room number.
- Don't change any column names. Each pair of students should only be shown once. The student whose name comes first in the alphabet should be shown first.
- A small hint: in terms of SQL, "first in the alphabet" means "smaller than" for text values.

SOLUTION

```
SELECT s1.name,
    s2.name,
    room_number
FROM std s1
JOIN std s2
    ON s1.room_id = s2.room_id
JOIN room
    ON s1.room_id = room.id
WHERE beds = 2
    AND s1.name < s2.name;
```

Subqueries

- We will now get to know subqueries. They are a powerful tool which will help you build complex instructions.
- Let’s go back to our HR user
- All right, to give you an idea of what subqueries are, consider the following problem: we want to find employees who work in the same department as ‘David Austin’
- We have two was to do that, can you explain one?

Subqueries

```
SELECT DEPARTMENT_ID
FROM EMPLOYEE    S
WHERE FIRST_NAME = 'David'
  AND LAST_NAME = 'Austin'
```

```
SELECT *
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 60;
```

```
SELECT *
FROM EMPLOYEES
WHERE DEPARTMENT_ID = (

  SELECT DEPARTMENT_ID
  FROM EMPLOYEES
  WHERE FIRST_NAME = 'David'
    AND LAST_NAME = 'Austin'

);
```

Subqueries with various logical operators

- Subqueries can also be used with other logical operators.
- Get all the employees who gets paid higher than David Austin.

```
SELECT *
FROM EMPLOYEES
WHERE SALARY > (
    SELECT SALARY
    FROM EMPLOYEES
    WHERE FIRST_NAME = 'David' AND LAST_NAME = 'Austin'
);
```

Subqueries with functions

- Select all the employees who gets paid less than the average.
-

```
SELECT *
FROM EMPLOYEES
WHERE SALARY < (
      SELECT AVG(SALARY)
      FROM EMPLOYEES
);
```

The operator IN

- So far, our subqueries only returned **single values**.
- To change that, we need to learn a new operator.

```
SELECT *
FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (90, 60, 100);
```

- Can you guess what this code does? Can we do the same thing without the ‘IN’ operator?

The operator IN with subqueries

- Can you list all of the employees in a department that has at least one employee whoose first name is David?

```
SELECT *
FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (
      SELECT DEPARTMENT_ID
      FROM EMPLOYEES
      WHERE FIRST_NAME = 'David'
);
```

The operator ALL

- = ALL
- != ALL
- < ALL
- <= ALL
- > ALL
- >= ALL
- Select all jobs which has a max salary greater than any employee

SOLUTION

```
SELECT *
FROM JOBS
WHERE MAX_SALARY > ALL (
      SELECT SALARY
      FROM EMPLOYEES
);
```

- ALL checks that it will achieve the condition **for each** object given after it

The operator ANY

- ANY works the same as ALL, expect that ANY requires just to achieve the condition on **at least one** object.

```
SELECT *
FROM JOBS
WHERE MAX_SALARY > ALL (
      SELECT SALARY
      FROM EMPLOYEES
);
```

INSERT INTO Statement

- The INSERT INTO statement is used to insert new records in a table.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Insert Multiple Rows

- It is also possible to insert multiple rows in one statement.
- To insert multiple rows of data, we use the same INSERT INTO statement, but with multiple values:
-

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES
(value1, value2, value3, …),
(value1, value2, value3, …),
(value1, value2, value3, …),
(value1, value2, value3, …);
```

The SQL UPDATE Statement

- The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- Note: Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

The SQL DELETE Statement

- The DELETE statement is used to delete existing records in a table.

```
DELETE FROM table_name WHERE condition;
```

- Note: Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

SQL Comments

- Comments are used to explain sections of SQL statements, or to prevent execution of SQL statements.

  - Single Line Comments
    - -- your comment

  - Multi-line Comments
    - /*
      your lo00o0o0o0o
      0o0ong comment
      */

SQL CREATE TABLE Statement

- The CREATE TABLE statement is used to create a new table in a database.

To create a new table

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

To create from existiong one

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

SQL TRUNCATE TABLE

- The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.
-

```
TRUNCATE TABLE table_name;
```

Delete a Table

- To delete the table completely, use the DROP TABLE statement:

```
DROP TABLE table_name;
```

ALTER TABLE Statement

- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- It’s also used to add and drop various constraints on an existing table.

ADD/DROP Column

- To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype;
```

- To delete a column in a table, use the following syntax

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

RENAME Column

- To rename a column in a table

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

- To rename a column in a table in SQL Server

```
EXEC sp_rename 'table_name.old_name',  'new_name', 'COLUMN';
```

ALTER/MODIFY DATATYPE

- To change the data type of a column in a table, use the following syntax:
  - My SQL / Oracle (prior version 10G):

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

  - Oracle 10G and later:

```
ALTER TABLE table_name
MODIFY column_name datatype;
```

  - SQL Server / MS Access:

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype;
```

.

SQL Constraints

- SQL constraints are used to specify rules for data in a table.

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

SQL Constraints

- The following constraints are commonly used in SQL:
  - NOT NULL - Ensures that a column cannot have a NULL value
  - UNIQUE - Ensures that all values in a column are different
  - PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
  - FOREIGN KEY - Prevents actions that would destroy links between tables
  - CHECK - Ensures that the values in a column satisfies a specific condition
  - DEFAULT - Sets a default value for a column if no value is specified
  - CREATE INDEX - Used to create and retrieve data from the database very quickly

NOT NULL Constraint

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.

UNIQUE Constraint

- The UNIQUE constraint ensures that all values in a column are different.

PRIMARY KEY Constraint

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).
-

FOREIGN KEY Constraint

- The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.
-

CHECK Constraint

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a column it will allow only certain values for this column.
-

The End

By: Trosc Team