SQL BASICS

1. Select Data from Table
2. Create Tables
3. SQL Functions

Database = organised collection of data, with data usually stored in tables.

- DB
    - Table
        - Data

Relational vs Non-Relational:

- Relational **(RDBMS)**:
    - Table based.
    - structured database, using columns and rows, with headers.
    - data is stored in a large amounts.
    - Access using SQL.
    - Eg. **MySQL, Oracle, Microsoft SQL Server**.
- Non-Relational **(DBMS)**:
    - document based.
    - unstructured database, using keywords:values.
    - data is stored in small amounts.
    - Eg. **MongoDB, Apache Cassandra**.

EPISODE ONE - How to select data from database

Show a Table:

```
DESCRIBE table_name;
```

Where operator:

- find items within a numeric range

```
Select *
FROM player
WHERE weight>190 AND height>200
or
WHERE weight>190 OR height>200;
```

- find items between a range

```
Select *
FROM player
WHERE weight BETWEEN 150 AND 200;
```

- find items by name

```
Select *
FROM player
WHERE player_name LIKE “Aaron Creswell”;
or start with string
WHERE player_name LIKE “Aaron%”;
```

or end with string
WHERE player_name LIKE “%Aaron”;
or starts with and ends
WHERE player_name LIKE “A%n”;
or one char = _ , so any string with T+char+m and then anything
WHERE player_name LIKE “T_m%”;

- find NULL values

Select *
FROM match
WHERE home_player_1 is null;
or
WHERE home_player_1 is not null;

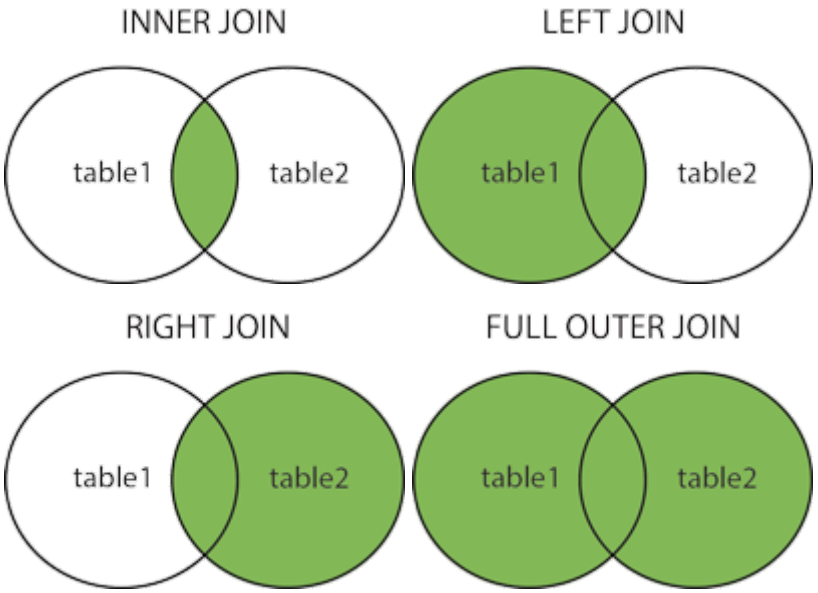
Sort (ORDER BY) operator:

- sort by ascending/descending order

Select *
FROM player
ORDER BY weight;
or
ORDER BY weight desc;

JOIN data from tables:

- Four Join Types:
  - INNER JOIN
  - LEFT (OUTER) JOIN
  - RIGHT (OUTER) JOIN
  - FULL (OUTER) JOIN



- Join two tables on two attributes in both tables (.player_api_id)

SELECT
player_attributes.player_api_id,

```
        player.player_name,
        player_attributes.date,
        player_attributes.overall_rating
    FROM
        Player_Attributes INNER JOIN Player ON
    Player_Attributes.player_api_id=Player.player_api_id;
```

- Clean up using *Alias (FROM player as p)*

```
    SELECT
        a.player_api_id,
        b.player_name,
        a.date,
        a.overall_rating
    FROM
        Player_Attributes a INNER JOIN Player b ON
    Player_Attributes.player_api_id=Player.player_api_id;
```

- SUM operator:
  - add up each players overall ratings - one per match
  - Group by id and name to have each players total overall rating
  - Remove .date attribute as we want total rating over all dates

```
    SELECT
        a.player_api_id,
        b.player_name,
        SUM(a.overall_rating) as rating
    FROM
        Player_Attributes a INNER JOIN Player b ON
    Player_Attributes.player_api_id=Player.player_api_id
    GROUP BY    a.player_api_id,
                b.player_name;
```

- AVG operator:

```
    ....
    AVG(a.overall_rating) as average_rating
    ....
```

- COUNT operator:

```
    ....
    COUNT(a.overall_rating) as num_ratings
    ....
```

- Now sort (ORDER) this by descending

```
    SELECT
        a.player_api_id,
        b.player_name,
        SUM(a.overall_rating) as rating
    FROM
        Player_Attributes a INNER JOIN Player b ON
    Player_Attributes.player_api_id=Player.player_api_id
```

```
GROUP BY    a.player_api_id,
            b.player_name
ORDER BY rating desc;
```

- HAVING operator:
  - only used with GROUP BY operator
  - must be immediately after GROUP BY
  - similar to WHERE but only for numeric searches

```
SELECT
        a.player_api_id,
        b.player_name,
        SUM(a.overall_rating) as rating
FROM
        Player_Attributes a INNER JOIN Player b ON
Player_Attributes.player_api_id=Player.player_api_id
GROUP BY
        a.player_api_id,
        b.player_name
HAVING
        rating>85
ORDER BY
        rating desc;
```

EPISODE TWO - Creating Tables

Syntax:

```
| CREATE TABLE table_name( |
|         field_name data_type, |
| );                          |
```

What is a Table:

- SQL database is made up of *tables*.
- Tables made up of *rows* and *columns*.
- Each ***column* or *field***: contains a *metric* or *dimension*
  - metric = quantitative numbers and measure stuff.
  - dimension = qualitative and describe stuff.
- Each ***row***: contains an *entry* in the table.

Common Data Types:

- **INT**
- **DECIMAL(s,d)** -s=size(number of digits),d=decimal places
  - [eg. decimal(6,2)]
- **VARCHAR(n)** - n=number of characters
- **DATE** - format= “YYYY-MM-DD”
- **TIMESTAMP** - format= “YYYY-MM-DD HH:MM:SS”

CREATE a Table:

```
CREATE TABLE bond(
    id INT,
    title VARCHAR(255),
    released INT,                    #not DATE bc we can use INT for calculations
    actor VARCHAR(255),
    director VARCHAR(255),
    box_office DECIMAL(10,2)
);
```

DELETE a Table:

```
DROP TABLE bond;
```

Constraints:

- **PRIMARY KEY** (most common):
  - recommended for every table
  - *uniquely* identifies every row (entry) in table
  - cannot contain NULL (empty) values
  - only ONE per table
  - usually use field that is already guarantees a unique value, for us thats id.

```
CREATE TABLE bond(
    id INT PRIMARY KEY,
    title VARCHAR(255),
    ….
);
```

- **Unique:**
  - no duplicates in a field
  - can contain single NULL value
  - can have more than one per table
  - For us we want only one entry per bond movie so use the title.

```
CREATE TABLE bond(
    id INT PRIMARY KEY,
    title VARCHAR(255) UNIQUE,
    ….
);
```

- **NOT NULL**
  - doesnt allow NULL (empty) value in the field for every single entry.
  - for us we can use this on *released* as each film cant not have been released

```
CREATE TABLE bond(
    id INT PRIMARY KEY,
    title VARCHAR(255) UNIQUE,
    released INT NOT NULL,
    ……
);
```

- **DEFAULT**
  - replaces NULL values with specified default value
  - if field is missing a value, we can auto replace it.4

- for us to say the newest film doesn't have box office data yet, we can set it to default.

```
CREATE TABLE bond(
        id INT PRIMARY KEY,
        title VARCHAR(255) UNIQUE,
        released INT NOT NULL,
        ……
        box_office DECIMAL(10,2) DEFAULT '0.0'   #must use single quotes
);
```

- Full TABLE now:

```
CREATE TABLE bond(
        id INT PRIMARY KEY,
        title VARCHAR(255) UNIQUE,
        released INT NOT NULL,
        actor VARCHAR(255),
        director VARCHAR(255),
        box_office DECIMAL(10,2) DEFAULT '0.0'
);
```

Insert row into SQL table:

```
INSERT INTO bond VALUES(
        1,
        'DR.NO',
        1962,
        'Sean Connery',
        'Terence Young',
        59.50
);
```

or

Insert data when you dont know all fields:

- fields will be filled with NULL or default values set earlier

```
INSERT INTO bond(id,title,released) VALUES(
        2,
        'From Russia With Love',
        1963
);
```

Delete a *row*:

```
DELETE FROM bond
WHERE title='Never Say Never Again';
```

Modify a Table:

- Add a *column* (**ALTER TABLE - ADD**)
    - add column for the studio

```
ALTER TABLE bond ADD studio VARCHAR(255);
```

- Delete a *column* (**ALTER TABLE - DROP**)

```
ALTER TABLE bond DROP studio;
```

- Change values (**UPDATE**)

```
UPDATE bond
SET actor='Connery'
WHERE actor='Sean Connery';
```

EPISODE THREE - SQL Functions

Using SQL functions to manipulate existing data and create new fields.
These new fields can be useful for:

- Grouping values
- Creating new values
- Cleaning data:
  - Correcting poorly formatted or incorrect data so that it can be analysed.

Data is usually corrupted when copying data multiple times, for example adding spaces before or after or in between strings.

4 claire gute
1 claire gute

- To us these values look the same, but to SQL the second one has a space at the end so they are handled as two separate values!!!

How to clean:

1. *TRIM* = remove extra spaces
   - **LTRIM(string)**
     - removes *leading* spaces
     - Our example: Table of sales in a USA store, has a column segment that has two values with one having a space error:
       - "Consumer"
       - "Consumer "

```
SELECT
        count(*) as count,
        TRIM(Segment) as segment_trim
FROM
        orders
GROUP BY
        segment_trim
```

   - **RTRIM(string)**
     - removes *trailing* spaces
   - **TRIM(string)**
     - removes *both leading and trailing* spaces

2. *LEFT/RIGHT* = returns specified number of characters
   - **LEFT(string, length)**

- returns specified number of characters from the *left* of the string
- Our example: Table of sales in a USA store, has a column order_id with values such as:
    - CA-2016-152412
    - US-2015-162223
    - create new column that only uses the first two characters (CA,US)

```
SELECT
        count(*) as count,
        TRIM(Segment) as segment_trim
        LEFT(order_id,2) as order_cat
FROM
        orders
GROUP BY
        segment_trim,
        order_cat;   #if its added to SELECT it must be in GROUP BY
```

- **RIGHT(string, length)**
    - returns specified number of characters from the *right* of the string

3. *PADDING* out column values:
    - **LPAD(str,len,padstr)**
        - Left pads a string with another string
        - Our Example: we have a zip_code column with all different lengths, if we wanted them all the same length using leading zeros.

```
SELECT
        zip_code,
        LPAD( zip_code, 5, '0')
FROM
        orders;
```

    - **RPAD(str,len,padstr)**
        - right pads a string with another string

4. Extract a substring from a value (commonly used):
    - **SUBSTRING(str, start_pos, len)**
        - Returns a specified number of characters from a particular position of a given string.
        - Our Example: our table has order_id like:
            - CA-2016-152156

        But we want to extract just the central four digits (2016)

```
SELECT
        SUBSTRING(order_id,4,4) as order_num     #starts at pos 4
FROM
        orders;
```

5. Return the length of a string (commonly used with SUBSTRING ^^^)
    - **LENGTH(str, pos, len)**
        - returns the length of a given string
        - can be used in SELECT or in WHERE

```
SELECT
        customer_name,
        LENGTH(customer_name)
FROM
        orders;

or
SELECT
        customer_name
FROM
        orders
WHERE
        LENGTH(customer_name)>10;
```

6. Return position of a substring
   - **LOCATE(substr, str)**
     - returns the position of the *first occurrence* of substring.
     - Our Example: return *first occurrence* of a space in a string

```
SELECT
        customer_name,
        LOCATE( ' ', customer_name )
FROM
        orders;
```

   - **POSITION(substr IN str)**
     - returns the position of substring within the string.

7. Use multiple operations together:
   - Our Example: all names in table are in format “John Smith” but we want to split this string into first and last name in format “John” and “Smith”
   - How?
     - firstname = go from Left first character to first space
     - lastname = go from first space to Right last character (len)
     - BUT we dont want the space, so we need -1 and +1

```
SELECT
        SUBSTR(customer_name, 1, LOCATE( ' ', customer_name)-1)
as fname,
        SUBSTR(customer_name, LOCATE( ' ', customer_name)+1,
LENGTH(customer_name)) as lname

FROM
        orders;
```

8. Change to Uppercase or LowerCase:
   - **UPPER(str)**
     - converts all characters in a string to uppercase

```
SELECT
        UPPER(customer_name)
FROM
```

```
        orders;
```

- **LOWER(str)**

9. Change just the first letter to Uppercase
    - Combination of a few operations
    - Combine LEFT, UPPER, LOWER, LENGTH functions
    - Then use CONCAT to reassemble
    - **CONCAT(str1, str2, …)**
        - used to add/join two or more strings

```
SELECT
        CONCAT(
                UPPER(LEFT( first_name, 1)),
                LOWER(SUBSTRING(first_name, 2,
                LENGTH(first_name)))
        ) AS new_firstname
FROM
(       # subquery from earlier ^^^
SELECT
        SUBSTR(customer_name, 1, LOCATE( ' ', customer_name)-1)
        as fname,
        SUBSTR(customer_name, LOCATE( ' ', customer_name)+1,
        LENGTH(customer_name)) as lname
FROM
        orders;
) AS names;
```

10. Conditional Expressions & Case Statements
    - Like IF-THEN-ELSE expressions
    - Can be used to group column values and fix errors like misspellings
    - Our Example: table has column *state* with value “Ohio” misspelt as “Ohios”
    - Fix with **CASE function**

```
SELECT
        state,
CASE
        WHEN state LIKE 'Ohios' THEN 'Ohio'
        ELSE
                state
        END
FROM
        orders;
```

11. More on Case statements
    - one CASE statements can have multiple WHEN statements
    - Group together rows in the product_name column into brands:
        - *Xerox, Acme, Avery*

```
SELECT
        product_name,
CASE
```

```
        WHEN product_name LIKE '%Xerox%' THEN 'Xerox'
        WHEN product_name LIKE '%Acme%' THEN 'Acme'
        WHEN product_name LIKE '%Avery%' THEN 'Avery'
        ELSE
            'Other
        END as brand
FROM
        orders;
```

----------END----------