Unit 5: SQL (queries)

- DML
- Queries
- Record filtering
- Null values
- Linking tables (*join*)
- Sub-queries and hierarchical queries (*in*)
- Alias
- Search through patterns: the *LIKE* operator
- Aggregate operators: *count, sum, avg, min, max, having*

DML

- DML → Data Manipulation Language
- It is the module within a SQL DBMS that allows to retrieve, input, modify or delete data
- We have now some data stored → let's see how to get them back from the database
- In SQL, this basically consists on the use of the statement SELECT

Queries

- Basic SQL query syntax:
- SELECT [DISTINCT] <attribute(s)> | <aggregate operator>(* | <attribute(s)>)

  FROM <table(s)>

  [WHERE <condition>]

  [GROUP BY <attribute list>]

  [ORDER BY <attribute list> [ASC | DESC]];

Queries

Some basic examples:

- SELECT * FROM drivers;
- SELECT dni, name FROM drivers;
- SELECT license_plate, dni FROM vehicles;
- SELECT DISTINCT color FROM vehicles;
- SELECT count(DISTINCT brand) FROM vehicles;

Queries

- In most of the cases, we will need to operate with more than one table at a time
- Feasible operations with tables (note that not all of them are part of the standard):
  - Union of tables
  - Intersection of tables
  - Difference of tables
  - Cartesian product of tables
  - Merging two tables by using common fields (*join*)

Operations on several tables

- Union:

  SELECT name FROM drivers

  UNION

  SELECT name FROM customers;

- Intersection:

  SELECT dni FROM drivers

  INTERSECT

  SELECT dni FROM vehicles;

Operations on several tables

- Difference:

  SELECT DNI FROM drivers

  EXCEPT -- or MINUS

  SELECT DNI FROM vehicles;
- Cartesian product (CROSS JOIN):

  SELECT * FROM drivers, vehicles;

Cartesian and natural product

- Through the cartesian product we can build combinations of values from two tables
- However, it *doesn't* actually connect those tables
- For the latter, we need the natural product (*join*)
- The natural product is not directly available, but we can use the cartesian product by filtering data to achieve the same result

Record filtering

- A query might return a large amount of data
- To avoid this, we must filter the data by putting conditions through predicates
- The logical condition that a query must fulfil is given through the clause WHERE
- In the condition we can make use of the logical operators: AND, OR, NOT, =, != (<>), >, <, >=, <=

Record filtering

- Examples:
- SELECT name, address

  FROM drivers

  WHERE name = 'John';
- SELECT dni

  FROM drivers

  WHERE license_date < '2000-01-01';

Record filtering

- Examples:
- SELECT license_plate

  FROM vehicles

  WHERE color = 'red'

  AND manufacturer = 'Mercedes';

Record filtering

- When a condition involves sets (in the form of tables), we can also use operators such as:
  - IN → data belonging to a set
  - ALL → every record in the set meets a condition
  - ANY → there is at least one record that fulfils the condition
- Some examples will be shown when sub-queries are brought up

Null values

- In the values of a certain table field, null values can be presented, as long as the NOT NULL clause is *not* used
- As we already know, a null value represents the absence of value in a field of a record (that is, an unknown or non-applicable value)
- To check if a value is null, we must use the IS NULL operator (never ~~= NULL~~) in the WHERE clause of the query

Null values

- Likewise, to get records with actual values in a given field, we must use IS NOT NULL (and not ~~<> NULL~~)
- Example:

```
SELECT dni, name
FROM employees
WHERE phone_number IS NULL;
```

Linking tables

- We have already seen some operations to work with more than one table
- However, the most common operation is to "connect" tables by common fields (typically, a foreign key)
- To achieve this, it is possible to make use the JOIN clause and its variants (INNER join, NATURAL join, LEFT join, RIGHT join)

Linking tables

- What does JOIN do? How does it work?
- To "link" or "merge" two tables through the JOIN operation, it is an essential condition that one of the tables has a foreign key to the other one
- This way, when applying JOIN, the SQL interpreter will connect both tables into one where the value of both fields matches

Linking tables

- The common columns will be called in the same way
- They will appear just once in the result table (since their values match)
- Example:

Table 'drivers'

dni	name

Table 'vehicles'

dni	brand	model

Linking tables

- Therefore, how do we link tables in SQL?
- Cartesian product:

  SELECT * FROM drivers, vehicles;
- Cartesian product with matching field values:

  SELECT * FROM drivers, vehicles

  WHERE drivers.dni = vehicles.dni;

Linking tables

- With NATURAL JOIN:

  SELECT * FROM drivers NATURAL JOIN vehicles;

- And in the event that the field names of the common data had different names:

  SELECT * FROM drivers INNER JOIN vehicles ON dni = driver_id;

Linking tables

- LEFT / RIGHT JOIN → these are special types of the JOIN operator to add some more rows in the result (either on the left or on the right)
- For example: we want to find out the data of all the drivers (whether they are owners or not) ...
- …and, in the case that they do own a vehicle, then append the data of this one to the result table

Linking tables

- The following query can be used for that purpose:
- SELECT *

  FROM drivers

  LEFT JOIN vehicles

  ON dni = dni_driver;

Linking tables

- As a result → the data of all of the drivers (and also their vehicles) are shown
- When a certain driver record does not own a car → a null value is present
- To restrict the result to those drivers with a vehicle → make use of the IS NOT NULL operation on the vehicle record values

Linking tables

- RIGHT JOIN is the other side version of LEFT JOIN
- By using it, we may rewrite the previous example the other way around
- Get the data of the vehicles and their drivers, and also add those ones who are not owners
- SELECT * FROM vehicles RIGHT JOIN drivers ON driver_id = dni;
- Like before, we can make use of the IS NOT NULL operation to restrict the result

Linking tables

- FULL OUTER JOIN is the full and combined version of LEFT JOIN and RIGHT JOIN
- This operation will find all combinations of records
- When they do not match, it will fill with null values to the left or right of the result
- To use this operation and make sense, the involved fields do not have to be primary / foreign keys
- For the above reason, it is an operation performed mainly in less usual circumstances

Sub-queries / hierarchical queries

- SQL has the possibility to nest queries → i.e. one query within another
- This is done in the same way as with blocks in a programming language
- This way, the result of a (sub or inner) query can be re-used to compute the result of another one

Sub-queries / hierarchical queries

- Example:

```
SELECT title, earnings
FROM movies
WHERE earnings > ALL (
    SELECT earnings FROM movies WHERE title = 'La vita è bella' OR title = 'Todo sobre mi madre'
);
```

Sub-queries / hierarchical queries

- Example (with IN operator):

```
SELECT title
FROM movies
WHERE director IN (
    SELECT director FROM movies WHERE title = 'La vita è bella' OR title = 'Todo sobre mi madre'
);
```

Subqueries and hierarchical queries

- Example (with EXISTS operator):

```
SELECT title, director
FROM movies
WHERE EXISTS (
    SELECT name FROM actors WHERE name = director
);
```

Alias

- When a list of fields is requested, it could become long should the * operator not be used
- Also, if any of those fields are used in a predicate and they have a long name → cumbersome
- Fix: use aliases of each field (at our choice) when performing a query

Alias

- Example:

  SELECT name AS n, birth_date AS bd

  FROM drivers;

- Alternatively (shorter):

  SELECT name n, birth_date bd

  FROM drivers;

Alias

- Oddly enough, declaring an alias for a field does not allow to use it within the WHERE clause
- For instance, the next query will not work, and it will result into a syntax error:

```
SELECT name AS n, birth_date AS bd
FROM drivers
WHERE bd > '2000-01-01';
```

Alias

- Instead of that, the following wrapping technique must be used to effectively use the alias in the condition:

```
SELECT * FROM
(
    SELECT name AS n, birth_date AS bd
    FROM drivers
)
WHERE bd > ‘2000-01-01’;
```

Alias

- We may as well leverage aliases for table names
- They are not only very useful, but essential when it is required to operate on the same table related to itself
- Example:

```
SELECT S1.name FROM students S1, students S2
WHERE S1.province = S2.province
AND S2.name = 'Pepe';
```

Search with patterns

- When filtering by text, the equality operator becomes the first tool
- However, it is not unusual having to handle wider conditions → for instance, all the names starting with 'A'
- To do this, there is a powerful operator → LIKE (or NOT LIKE)
- The LIKE operator is used with wildcards: %, _

LIKE operator

- Examples:
- SELECT first_name, last_name FROM drivers WHERE first_name LIKE 'A%';
- SELECT first_name, last_name FROM drivers WHERE last_name LIKE '%z';
- SELECT first_name, last_name FROM drivers WHERE last_name NOT LIKE '%García%';

LIKE operator

- Examples:
- SELECT first_name, last_name FROM drivers WHERE first_name LIKE 'Mar_ _';
- SELECT first_name, last_name FROM drivers WHERE first_name LIKE '_ _ _ _ _';
- SELECT first_name, last_name FROM drivers WHERE last_name LIKE '_a%ez';

Aggregate operators

- There are a number of SQL operators to carry out certain computationsapplied to subsets of tuples
- These are the so called aggregate operators, and they are:
  - count
  - sum
  - avg
  - min
  - max
  - HAVING → clause used to verify conditions on the previous ones

Aggregate operators

- Examples:

  SELECT count(*) FROM drivers;

  SELECT count(id) FROM drivers WHERE name LIKE '%García%';

- The second syntax is preferable as avoiding the wildcard also reduces the computation time

Aggregate operators

- More examples:

  SELECT sum(earnings) FROM movies WHERE country = 'Spain';

  SELECT max(earnings) FROM movies WHERE director = 'Pedro Almodóvar';

  SELECT min(earnings) FROM movies WHERE release_date >= '2021-01-01';

Grouping results

- Aggregate operators are often used hand in hand with grouping the requested data
- Grouping consists on classify the results in groups according to given criteria → namely, the fields we wish to group by
- The GROUP BY clause followed by a list of attributes is used for that purpose
- This clause must be written after the WHERE clause
- When the GROUP BY clause is used, all of the attributes requested in the SELECT clause must be used BUT one of them → the groups will be applied in the same order

Grouping results

- Example:

```
SELECT number_plate, first_name, last_name
FROM drivers INNER JOIN vehicles
ON drivers.id = vehicles.driver_id
WHERE license_date >= ‘2008-01-01’
GROUP BY last_name, first_name;
```

Conditions on aggregates

- To apply conditions to the aggregation results, WHERE cannot be used
- To restrict the results to the calculations, the HAVING operator must be used along with with data sets
- Example:

  SELECT director, avg(earnings) FROM movies GROUP BY director HAVING avg(earnings) > '1000000';

Sorting results

- As known, the rows from relational model tables are not ordered
- However, SQL provide us with a way to request the results sorted
- In order to do that, the ORDER BY clause must be used after all the others (WHERE → GROUP BY → HAVING → ORDER BY)

Sorting results

- The ORDER BY clause works with at least one field, but we can add more to
- After the attribute, either the ASC or DESC modifiers can be used to modify the order (from low to high, or the other way around
- The ASC / DESC modifiers are optional → by default, the results are returned in ASCendent order when the ORDER BY clause is used

Sorting results

- Example:

  SELECT director, avg(earnings) AS avg_director

  FROM movies

  GROUP BY director

  HAVING avg(earnings) > '1000000'

  ORDER BY avg_director DESC, director ASC;