Lab III. Part A

SQL: database manipulation

Database Laboratory

Lab 3. Part A: SQL Data Manipulation

Objectives:

- To present the syntax of the Structured Query Language (only the manipulation part, DML).
- To see some simple examples to clarify the semantics of SQL.
- To present the databases CICLISMO, MÚSICA and BIBLIOTECA.
- To perform queries in SQL over these databases from easy to ever more complex queries.
- To perform all of this with the tool SQL Worksheet using the ORACLE DBMS.

SQL DML

We will present the instructions which can be executed from a SQL interpreter or console. This is called *interactive* SQL.

SQL is a very expressive language and, in general, allows the user to express the same command in many different ways.

The four instructions that compose the DML of SQL are:

- **select**: it allows the declaration of queries to retrieve the information from one or more tables in the database.
- **insert**: it performs the insertion of one or more rows in a table.
- **delete**: it allows the user to delete one or more rows from a table.
- **update**: it modifies the values of one or more columns and/or one or more rows in a table.

Queries: the SELECT command

select [**all** | **distinct**] *selected_item_commalist* | *

from *table_reference_commalist*

[**where** *conditional_expression*]

[**group by** *column_reference_commalist*]

[**having** *conditional_expression*]

[**order by** *column_reference_commalist*]

- *selected_item_commalist*: information we want to obtain from the database.
- **from** *table_reference_commalist*: it specifies from which tables we obtain the required information.
- **where** *conditional_expression*: it expresses a condition that the recovered rows must fulfil.
- **group by** *column_reference_commalist*: it allows the user to perform grouped queries to extract information from the formed groups.
- **having** *conditional_expression*: condition over the formed groups.
- **order by** *column_reference_commalist*: it sorts out data by one or more columns.

Conditions for simple queries

3 **select** [**all** | **distinct**] *selected_item_commalist* | *

1 **from** table

2 [**where** *conditional_expression*]

4 [**order by** *column_reference_commalist*]

- **all** : it allows identical rows to appear in the result (default value).
- **distinct**: it doesn't allow repeated rows in the result.
- The *conditional_expression* is composed of a set of predicates which are combined with the logical connectives **and**, **or** and **not**. The predicates that can be used are:
  - comparison predicates: =, <>, >, <, >=, <=.
  - **like**: it allows the comparison of a string with a pattern.
  - **between**: it allows checking whether a number is within a range.
  - **in**: it allows checking whether the value is within a set.
  - **is null**: it allows checking whether the value is null.

Examples

Example: Obtain the name and the age of all the “cyclists” (cycling racers).

```
SELECT nombre, edad FROM Ciclista;
```

Example: Obtain the name and the height of all the mountain passes (“puertos”) of category 1.

```
SELECT nompuerto, altura FROM Puerto
WHERE categoria = ‘1’;
```

Example: Obtain the name of the cyclists whose age is between 20 and 30.

```
SELECT nombre FROM Ciclista
WHERE edad BETWEEN 20 AND 30;
```

(*) The predicate “between” is equivalent to a condition such as:

exp between exp_1 and exp_2 $\equiv$ (exp >= exp_1) and (exp <= exp_2)

Examples

Example: Obtain the number of stages where the name of the arrival city (“llegada”) has an “O” as a second letter or where the departure city (“salida”) has two or more ‘A’s.

```
SELECT netapa FROM Etapa
WHERE llegada LIKE ‘_O%’ OR salida LIKE ‘%A%A%’;
```

Example: Obtain the name of the mountain pass of categories 1, 2 or 3.

```
SELECT nompuerto FROM Puerto
WHERE categoria IN ( ‘1’, ‘2’, ‘3’ ) ;
```

(*) The predicate “in” can also be expressed as:

exp in $(exp_1, exp_2, \ldots, exp_n) \equiv (exp=exp_1)$ or $(exp=exp_2)$ or…or $(exp=exp_n)$

Example: Obtain the data of the cyclists whose age we don’t know.

```
SELECT * FROM Ciclista
WHERE edad IS NULL;
```

Comparison of null values

The comparison between any value and NULL is *undefined*. Example:

```
select *
from T
where attrib1 > attrib2;
```

If in a row we have that $attrib_1$ = 50 and $attrib_2$ is null, the result of the comparison will be undefined and this row won't be included in the selection.

Example of an incorrect query (syntax error)

```
select nomeq
from Equipo
where  director = null;
```

More Examples

Use of arithmetic operators: + (sum), – (difference), * (product), / (quotient), etc.

Example: Obtain the type and prize in euros (assuming that it is in pesetas in the database) of those jerseys whose prize is greater than 100 euros.

```
SELECT tipo, premio / 166.386 FROM Maillot

WHERE premio / 166.386 > 100;
```

Use of LIKE

Example: Obtain the name and age of the cyclists who belong to the teams whose name contains the string "100%".

```
SELECT nombre, edad FROM Ciclista

WHERE nomeq LIKE ‘%100\%%’ ESCAPE ‘\’;
```

Queries with aggregated values

The syntax of the aggregated functions is as follows:

{ **avg** | **max** | **min** | **sum** | **count** } ([**all** | **distinct**] *scalar_expression*) | **count(*)**

- Aggregated functions cannot be nested.
- For the functions **sum** and **avg** the arguments must be numerical.
- **distinct** indicates that the redundant values will be removed before performing the calculus.
- The calculi are performed after the conditions in the WHERE clause.
- The null values are removed before performing the calculi (incl. **count**).
- If the number of rows in the selection is 0, the function **count** returns the value 0 and the rest return the null value.
- The special function **count(*)**, where we cannot place "distinct" or "all", gives the cardinality of the set of rows given by the WHERE clause.

Aggregated functions in non-grouped queries

Example:

```
SELECT ‘Núm. de ciclistas =’, COUNT(*), ‘Media Edad =’, AVG(edad)
FROM Ciclista
WHERE nomeq = ‘Banesto’;
```

In non-grouped queries, the SELECT clause can only include references to aggregated functions or literals, since the functions will return just a single value.

Incorrect example:

```
SELECT nombre, AVG(edad)
FROM Ciclista
WHERE nomeq = ‘ONCE’;
```

Simple queries over several tables

When the information we want to obtain is stored in more than one table, then it is necessary to declare a query which manipulates several tables.

Example: Obtain pairs of numbers of stages and names of mountain passes which have been won by the same cyclist.

```
select etapa.netapa, nompuerto
from Etapa, Puerto
where etapa.dorsal = puerto.dorsal;
```

In this expression it is compulsory that the reference to the column *dorsal* in *Etapa* and *Puerto* is qualified with the name of the table. Otherwise, it is ambiguous. In general,

[table | *run_variable*].*columna*

The "run variables" make it possible to give an alternative name (an alias) to the same table. A "run variable" is defined as follows:

from table [as] *run_variable*

Use of foreign keys for queries over several tables

A query over several tables corresponds to the Cartesian product.

- If we do not express several conditions to connect them, the number of rows will be very high.
- If there are foreign keys defined, it is usual that some conditions are formed by an equality between the foreign key and the corresponding attributes in the table to which it refers.

Example: Obtain the name of the cyclists who belong to the team coached by ‘Alvaro Pino’.

```
SELECT C.nombre FROM Ciclista C, Equipo E
WHERE C.nomeq = E.nomeq AND E.director = ‘Alvaro Pino’;
```

Example: Obtain the name of the cyclists and the number of stages such that the cyclist has won the stage. Additionally, the stage must be more than 150 km. long.

```
SELECT C.nombre, E.netapa FROM Ciclista C, Etapa E
WHERE C.dorsal = E.dorsal AND E.km > 150;
```

Complex queries: subqueries

If the required information is included in a table and the search condition for this information must be applied to other tables, then (in some cases) we can use subqueries to express this kind of conditions.

EXAMPLE: The previous example: Obtain the names of the cyclists who belong to the team coached by ‘Alvaro Pino’. We solved it through equalities:

```
SELECT C.nombre FROM Ciclista C, Equipo E

WHERE C.nomeq = E.nomeq AND E.director = ‘Alvaro Pino’;
```

And using subqueries:

```
SELECT C.nombre FROM Ciclista C

WHERE C.nomeq = (SELECT E.nomeq FROM Equipo E

                 WHERE E.director = ‘Alvaro Pino’);
```

This is possible since the information which is required, name of the cyclist, is not in the table used in the subquery (Equipo), and the subquery returns one single value.

Predicates which accept subqueries

Subqueries can appear in the search conditions, either in the “where” or in the “having” clause, as arguments of some predicates.

The predicates which can have a subquery as an argument are the following:

- Comparison predicates (=, <>, >, <, >=, <=).
- in: it checks that a value belongs to the collection (table) returned by the subquery.
- match: it checks whether a value is identical to a value in a collection.
- Comparison predicates with quantifiers (*any* and *all*): they allow the comparison of a value with a set of values.
- exists: it is equivalent to the existential quantifier; it checks whether a subquery returns some row.
- unique: it returns true if the query has no repeated rows.

Comparison Predicates (=,<>, >, <, >=, <=)

Each of the two sides of a comparison predicate must be a single tuple which must be formed by the same number of rows:

(A1, A2, …, An) comparison_predicate (B1, B2, …, Bn)

EXAMPLES:

(‘Pepe’, 28) <= (‘Juanito’, 39)

It will be false since ‘Pepe’ goes before ‘Juanito’ alphabetically

(‘Pepe’, 28) <= (‘Pepe’, 39)

It will be true since ‘Pepe’ = ‘Pepe’ and 28 < 39.

This is called the “lexicographic” extension of the comparison operators.

Comparison Predicates (=,<>, >, <, >=, <=)

Subqueries can be on any side (normally on the right) if and only if they return a single row and the number of columns match (in number and type) with the other half of the comparison predicate:

row_constructor comparison_predicate *row_constructor*

Where "row_constructor" is either a sequence of constants or a subquery. Example:

```
('Pepe', 28) <= (SELECT name, age FROM student where student_id = 7777);
```

- In the case the subquery is empty, the row is converted to a row with several null values and the result of the comparison will be undefined.
- When *row_constructor* returns more than one column, the lexicographic order will be used in the comparison of each operator (=, <>, >, <, >=, <=).
  - For simplicity, we will only see queries with one column in the subquery.

Comparison Predicates (=,<>, >, <, >=, <=)

EXAMPLE: Obtain the name of the mountain passes whose height is greater than the mean of the height of all 2nd category mountain passes.

```
SELECT nompuerto FROM Puerto
WHERE altura > (SELECT AVG(altura) FROM Puerto
                WHERE categoría = 2 );
```

INCORRECT: (execution error):

```
SELECT nompuerto FROM Puerto
WHERE altura > (SELECT altura FROM Puerto
                WHERE categoría = 2 );
```

IN predicate

```
row_constructor [not] in (table_expression)
```

On the righthand part of the IN predicate we can now put something (that we call “table_expression”) which returns more than a row.

EXAMPLE: Obtain the “netapa” of the stages which have been won by cyclists whose age is greater than 30 years.

```
SELECT netapa FROM Etapa
WHERE dorsal IN (SELECT dorsal FROM Ciclista
                 WHERE edad > 30);
```

Nested Queries

EXAMPLE: Obtain the number of stages won by cyclists who belong to teams whose coach (director) has a name beginning with ‘A’.

```
SELECT netapa FROM Etapa
WHERE dorsal IN (SELECT dorsal FROM Ciclista
WHERE nomeq IN (SELECT nomeq FROM Equipo
    WHERE director LIKE ‘A%’));
```