Lab. 05: SQL Instructions II

Introduction to Software Engineering and Information Systems I

Course 2021/22

David Ruiz, Inma Hernández, Agustín Borrego, Daniel Ayala

Index

1. Objective

The goal of this lab is to perform advanced SELECT queries that require new commands. The student will learn to:

- Use ORDER BY, LIMIT, and OFFSET to return ordered, limited, and paged records.
- Use JOIN to retrieve columns from multiple tables.
- Use GROUP BY to group rows.

2. Environment setup

Connect to the "degrees" database and run the tables.sql and populate.sql scripts.

Create a file queries-2.sql for writing the queries.

3. ORDER BY, LIMIT, and OFFSET

The records resulting from executing queries were sorted by insertion order so far. If we want to sort them differently the clause ORDER BY must be used. We obtain the grades ordered by score (from lowest to highest) as follows:

```
1 SELECT *
2    FROM Grades
3    ORDER BY value;
```

We can build more sophisticated queries. For example, we can retrieve the passed grades, ordered by the surname of the student who obtained them in reverse order:

```
5 SELECT *
6    FROM Grades
7    WHERE VALUE >5
8    ORDER BY (SELECT surname
9                 FROM Students
10                WHERE Students.studentId = Grades.studentId)
11   DESC;
```


Observe the following:

- ORDER BY is placed after WHERE, if present. If not, ORDER BY is used in its place.
- Any expression applicable to a single record may be used in the ORDER BY clause, even subqueries that return a value.
- The order is ascending (ASC) by default. If we want it to be descending, we have to employ the DESC clause.

In many cases, it is not necessary to obtain all the rows resulting from a query, but only a few (for example, because a page is displayed with only a few results). To limit the results obtained, we use LIMIT. We get the top 5 marks as follows:

```
14 SELECT *
15    FROM Grades
16    ORDER BY VALUE DESC
17    LIMIT 5;
```

We can also indicate that we want the first rows from a position. For example, if we want to paginate the grades, and we want to get the second page of 5 grades, we use the following query:

```
19 SELECT *
20    FROM Grades
21    ORDER BY VALUE DESC
22    LIMIT 5 OFFSET 5;
```

Notice how the value of OFFSET does not represent a page number, but the number of rows from which results will begin to be retrieved. If we wanted the third page (with 5 grades per page), the value of OFFSET would be 10.

4.JOIN

Until now, SELECT queries have involved only one table (placed in the FROM clause). If we write several tables, separated by commas, the Cartesian product is obtained, that is, all the possible combinations of rows between the tables:

```
23 SELECT *
24    FROM Groups, GroupsStudents, Students;
```

Run the previous query. The result contains all the columns of the tables and 6048 rows, corresponding to all the possible record combinations of each table. We usually want to combine records which share the value of some specific colum , one that is a primary key in one table and foreign key in the other table. In this way we can expand the resulting set of columns of one table with columns of another one.

To obtain the union of rows of different tables by one or more attributes, we use JOIN:

```
26 SELECT *
27    FROM Groups
28    JOIN GroupsStudents ON (Groups.groupId = GroupsStudents.groupId)
29    JOIN Students ON (GroupsStudents.studentId = Students.studentId);
```

Observe the following:

- We indicate the tables to be joined, followed by how the records will be joined.
- As there are several tables with columns with the same name, we must always indicate the table that comprises that specific column.
- Multiple tables can be joined at the same time.
- If a group has no students, it will not appear. Similarly, if a student does not have groups, they will not appear. JOIN variants can be used to return all rows from one table adding, if any, information from the other. They are the LEFT JOIN, RIGHT JOIN clauses and similar ones, which can be consulted in the MariaDB page.

When two tables are to be joined by the value of one specific column, and that column has the same name in both tables, the NATURAL JOIN clause can be used instead of specifying manually the join condition, which performs the operation automatically. This is especially useful for joining two or more tables using foreign key relationships if they have the same name in both related tables.

So the above query can be written as:

```
1 SELECT *
2    FROM Groups
3    NATURAL JOIN GroupsStudents
4    NATURAL JOIN Students;
```

5. GROUP BY

Some of the queries written so far obtained aggregated values by using commands such as MAX, COUNT, or AVG. In this way, we could get a single maximum, minimum, etc. However, it may be necessary to obtain several aggregate measures corresponding to several groups. For this we may employ the GROUP BY clause. To obtain the average grade of each student together with their name and surname, we can execute the following query:

```
37 SELECT firstName, surname, AVG(value)
38    FROM Students
39    JOIN Grades ON (Students.studentId = Grades.studentId)
40    GROUP BY Students.studentId;
```

Observe the following:

- We indicate with GROUP BY the criteria by which to form groups.
- The attributes to be selected will be selected for each group. The attributes that don't belong to the aggregation, but to a single record, will be taken from the first row of the group.
- Since we know that the first and last names of the students in each group of rows are the same (group = result after grouping by student id), it is correct to ask for them. Otherwise, the firstName and surname of the first resulting record would be retrieved.

Next, we make a query to obtain the average grade in each call for each subject in 2018:

```
42 CREATE OR REPLACE VIEW ViewSubjectGrades AS
43    SELECT Students.studentId, Students.firstName, Students.surname,
44           Subjects.subjectId, Subjects.name,
45           Grades.value, Grades.gradeCall,
46           Groups.year
47    FROM Students
48    JOIN Grades ON (Students.studentId = Grades.studentId)
49    JOIN Groups ON (Grades.groupId = Groups.groupId)
50    JOIN Subjects ON (Groups.subjectId = Subjects.subjectId);
51
52 SELECT gradeCall, name, AVG(value)
53    FROM ViewSubjectGrades
54    WHERE VALUE >= 5 AND year = 2018
55    GROUP BY gradeCall, subjectId;
```

Observe the following:

- There is no table that contains the marks of each student in each subject, since the marks are associated with groups, and the groups with subjects. Instead of creating a long query, we here decided to first create a view that gathers the information from several tables to comfortably have the marks of each student in each subject. The creation of a view is not required.
- We can filter the rows before grouping them.
- We can group according to two criteria, so that the groups will be created with rows that have the same value in all the specified attributes.

Finally, we obtain the average mark of the subjects with more than 2 marks (in all years):

```
57 SELECT name, AVG(VALUE)
58    FROM ViewSubjectGrades
59    GROUP BY NAME
60    HAVING COUNT(*) > 2;
```

Notice how we filter the groups with a condition in HAVING. Do not confuse it with WHERE, which filters the rows before grouping them.

6. Various queries

- Number of students born in each year.

```
64 SELECT YEAR(birthdate), COUNT(*)
65    FROM Students
66    GROUP BY YEAR(birthdate);
```

- Number of students per degree in the 2019 academic year.

```
64 -- Vista con los estudiantes de cada grado
65 CREATE OR REPLACE VIEW ViewDegreeStudents AS
66    SELECT Students.*, Degrees.*, year
67    FROM Students
68    JOIN GroupsStudents ON (Students.studentId = GroupsStudents.studentId)
69    JOIN Groups ON (GroupsStudents.groupId = Groups.groupId)
70    JOIN Subjects ON (Groups.subjectId = Subjects.subjectId)
71    JOIN Degrees ON (Subjects.degreeId = Degrees.degreeId);
72
73 SELECT name, COUNT(*)
74    FROM ViewDegreeStudents
75    WHERE year = 2019
76    GROUP BY NAME;
```


- Maximum mark of each student, with the name and surname.

```
81 SELECT firstName, surname, MAX(VALUE)
82    FROM ViewSubjectGrades
83    GROUP BY studentId;
```

- Name and number of theory groups of the 3 subjects with the highest number of theory groups in 2019.

```
86 -- Vista con las asignaturas de cada grupo
87 CREATE OR REPLACE VIEW ViewSubjectGroups AS
88    SELECT Subjects.*, Groups.name AS groupName, Groups.activity, Groups.year
89    FROM Subjects JOIN Groups ON (Subjects.subjectId = Groups.subjectId);
90
91 SELECT name, COUNT(*)
92    FROM ViewSubjectGroups
93    WHERE year = 2019 AND activity = 'Teoria'
94    GROUP BY subjectId
95    ORDER BY COUNT(*) DESC LIMIT 3;
```

- Name and surname of students per year who had an average grade higher than the average grade for the year. Notice that two views are employed here.

```
 98 -- Vista con la nota media de cada año
 99 CREATE OR REPLACE VIEW ViewAvgGradesYear AS
100    SELECT year, AVG(VALUE) AS average
101    FROM ViewSubjectGrades
102    GROUP BY year;
103
104 SELECT firstName, surname, year, AVG(VALUE) AS studentAverage
105    FROM ViewSubjectGrades
106    GROUP BY studentId, year
107    HAVING (studentAverage > (SELECT average
108                              FROM ViewAvgGradesYear
109                              WHERE ViewAvgGradesYear.year = ViewSubjectGrades.year));
```

- Name of subjects that belong to a degree with more than 4 subjects.

```
112 -- Vista con el número de asignaturas de cada grado
113 CREATE OR REPLACE VIEW ViewDegreeNumSubjects AS
114    SELECT Degrees.degreeId, COUNT(*) AS numSubjects
115    FROM Subjects JOIN Degrees ON (Subjects.degreeId = Degrees.degreeId)
116    GROUP BY degreeId;
117
118 SELECT name
119    FROM Subjects
120    WHERE (SELECT numSubjects
121           FROM ViewDegreeNumSubjects
122           WHERE ViewDegreeNumSubjects.degreeId = Subjects.degreeId) > 4;
```


7.Exercises

Implement the queries that retrieve the following data. Create the views you consider adequate:

- Number of failures of each student, given the name and surname.
- The third page (each page contains 3 groups), sorted by year in descending order.
- A list of the groups, adding the acronym of the related subject and the name of the related degree.
- Number of different access methods of the students in each group, given the group id.
- Grade weighted by credits of each student, giving name and surname, of the 2019 course in the first call. Hint: Modify the ViewSubjectGrades view by adding the missing attribute. The weighted grade is equal to the sum of each grade multiplied by the credits of its subject, divided by the sum of all the credits of the subjects.

To do the exercise, create a copy of the repository for this session with your GitHub user. Make appropriate modifications to the supplied files, creating new files if necessary, and upload those changes to your copy on GitHub. Remember to set the privacy of your repository as "Private" and give dfernandez6 user access as a collaborator.