Hand-In 2

- Stefan Jacobsen, Thomas Richardy, Jacob Hartmann, Villads Nielsen

1.

We have specified our database in a DDL-file, which is to be found in the main directory. We chose this to collect all of the creations of the tables in one file, and to follow what has been noted in the lectures as the common standard for data definitions.

If we are to dig deeper into how we have structured the code**,** we followed the same reasoning as we followed with our E/R diagram from the first hand-in, where our focus was on our three main entities, Student, Course and Project, which always will be the three most important tables. The following six tables are all of our supporting tables, which inherit the most, and collect a lot of the specific information which is shared between the tables.

To explain some of the functional parts, we have chosen to rely heavily on the FOREIGN KEY Constraint, which means that we can inherit the different Primary Keys and Keys, from their original tables to the ones in which we need the inherited values. This is a bit difference from how we chose to do it in our diagram, since by doing this, we can cut down on the amount of entities we actually implement. The reasoning behind doing it like this, was that we thought it played better to the strengths of SQL.

Following some of our table creation, we have added a Create Index statement, this is in the hope that we might be able to speed up the matching between tables, and more specifically for bug-killing.

In our DML file, you will find the queries we are asked to make in the next couple of questions.

2.

Data has been generated using regular expression in Ruby, and some fantastic Google-magic. This has been done to make the query-results as non-trivial as possible. The entire datasets are to be found in the Data-directory.

More specifically, we have generated 30 projects, 50 courses, 75 teachers(and thereby also 75 supervisors), 200 individual students and 1000 Hand-Ins. Each of them should be specific and individual in his or her’s characteristics. But since we have a lot of relationships in our model, there is a lot of shared information between the tables.

3.

In this section we offer a solution to each query-problem, and provide reasoning behind our code, as we go along.

Which students have received a grade that is 10 or lower?

```
SELECT Student_Name AS name, Student.Student_id AS ID
        FROM Student, StudentDoesProject, Exam
        WHERE StudentDoesProject.Student_id = Student.Student_id
              AND Exam.Student_id = Student.Student_id
              AND (Project_Grade <= 10 OR Exam_Grade <= 10);
```

Reasoning for Query

The idea behind this query, is to use the 'WHERE' clause to make sure that all grades we query, are connected to the correct student, and get both the project grades and the exam grades. Using the AND operator and arithmetic operators seem to do the job. In the select-clause we rename columns from readability and aesthetic reasons.

Which teachers have taught the same course?

```
SELECT Teacher_Id, Course_Id
        FROM TeacherTeachesCourse ttc1, TeacherTeachesCourse ttc2
              WHERE ttc1.Course_Id = ttc2.Cource_Id
```

Reasoning for Query

To match teachers against themselves, we create two aliases of the same table, to match attributes of this table against themselves in the WHERE clause. The key to this approach is that we have a table that matches the relationship between the Student table and the Teacher table.

List each student together with the average grade for the student across all courses the student has taken.

```
SELECT Student_id, AVG(Exam_Grade)
        FROM Exam
            GROUP BY Student_id
```

Reasoning for Query

This query builds towards the following query, so the most important idea, is to use the AVG operator to get an average of grades per student, and identify the students as unique by their Id.

How many distinct average grades are there among students?

```
SELECT COUNT(DISTINCT (Average)) AS NumOfAvgGrades
    FROM (SELECT Student.Student_id, AVG(Exam_Grade) AS Average
            FROM Student, Exam)
```

Reasoning for Query

For this query, we opted to insert a subquery inside the outermost from-clause, in order to have the average grade per student, while in the outermost query only counting the distinct averages. Notice how we are able to reuse the query from the previous query. Again we rename the resulting column into a slightly more sensible name, as, although the query-code is relatively simple, the data requested is starting to become somewhat of a mouthful.

The output from each query has been included below.