UniversityDB Project

Application Domain

PROBLEM STATEMENT

We want to be able to represent some important data of a university in a structured and organized way, possibly to lay a foundation for building an interactive website for current students and prospective students to use, but also for the purpose of storing and analysing our data or automating the generation of new data. We consider the following to be important:

1. information about professors, assistants, courses and students
2. information about attendance and exam (for simplicity and performance we store only current semester's information)
3. storing and analysing surveys from students in the end of the semester

Now let's move on to the next step in our software development activity.

REQUIREMENTS ELICITATION

Developing our project is a case of greenfield engineering, meaning that no prior system exists and we will develop from scratch. In this case, requirements are extracted from the client and the user. Since I am the client in this case, I provided my own requirements for the project and took into consideration what a possible user of our project might have as requirements too.

Architectural Requirements

We choose to use MySQL as the SQL dialect and our database application to align with the use of this database in the lecture, but also try to use as little as possible MySQL specific code, in order to make possible future migrations arising from architectural changes as easy as possible.

Structural requirements

We propose that our project follows the following structural guidelines:
1. professors, assistants, courses and students must be their own entity
2. attendance and exam must be their own many to many relationship and as a result be their own associative entity.

3. we also propose an entity for storing requirements of courses in tuples (predecessor, successor)
4. survey must be a weak entity dependent on attendance.
5. we also propose a weak entity dependent on professors entity for storing the bonus of professors for that semester.

Functional Requirements

A functional requirement is a specification of a function that the system must support. Functional requirements include but are not restricted to: relationship of outputs and inputs, response to abnormal situations, exact sequence of operations and validity checks on the inputs. Because database systems allow for a multitude of possible functions given a sound structural representation of data by simply writing a new query, we can simply say that the functional requirements are already satisfied by our database vendor. We can execute various select, create, update, delete queries as we please in the future when the need arises.

Non-functional requirements

Non-functional requirements include performance requirements, quality requirements and pseudo Requirements (constraints).

1. Performance Requirements

very good performance in speed, response time and recovery time. Performance requirements are mostly dependent and fulfilled by the database vendor. Usually all functionalities that database systems are capable of perform excellently.
On our part we introduce the following constraint on associative entities like attend and exam: they only store current semester's information. This will boost performance since most queries are about the current semester. We also suggest that in the end of the semester, this information gets backed up in another table in the database like: attend_summer_semester_2020 and exam_summer_semester_2020, or maybe in a bigger attend and exam table containing information about all earlier semesters.

2. Quality Requirements

reliable as in robust, safe and secure, have high usability and supportability (adaptability, maintainability and portability).
Example for robustness: the system must be able to prevent unwanted delete queries that might violate integrity of the data.
Example of safety: Protection against unwanted incidents.
Example of security: Protection against intended incidents (hacks / intrusions)
Example of adaptability: the ability to adapt the behaviour given different data.
Example of maintainability: changes in the code from the developer to prevent bugs, correct behaviour or change features must be simple to do.

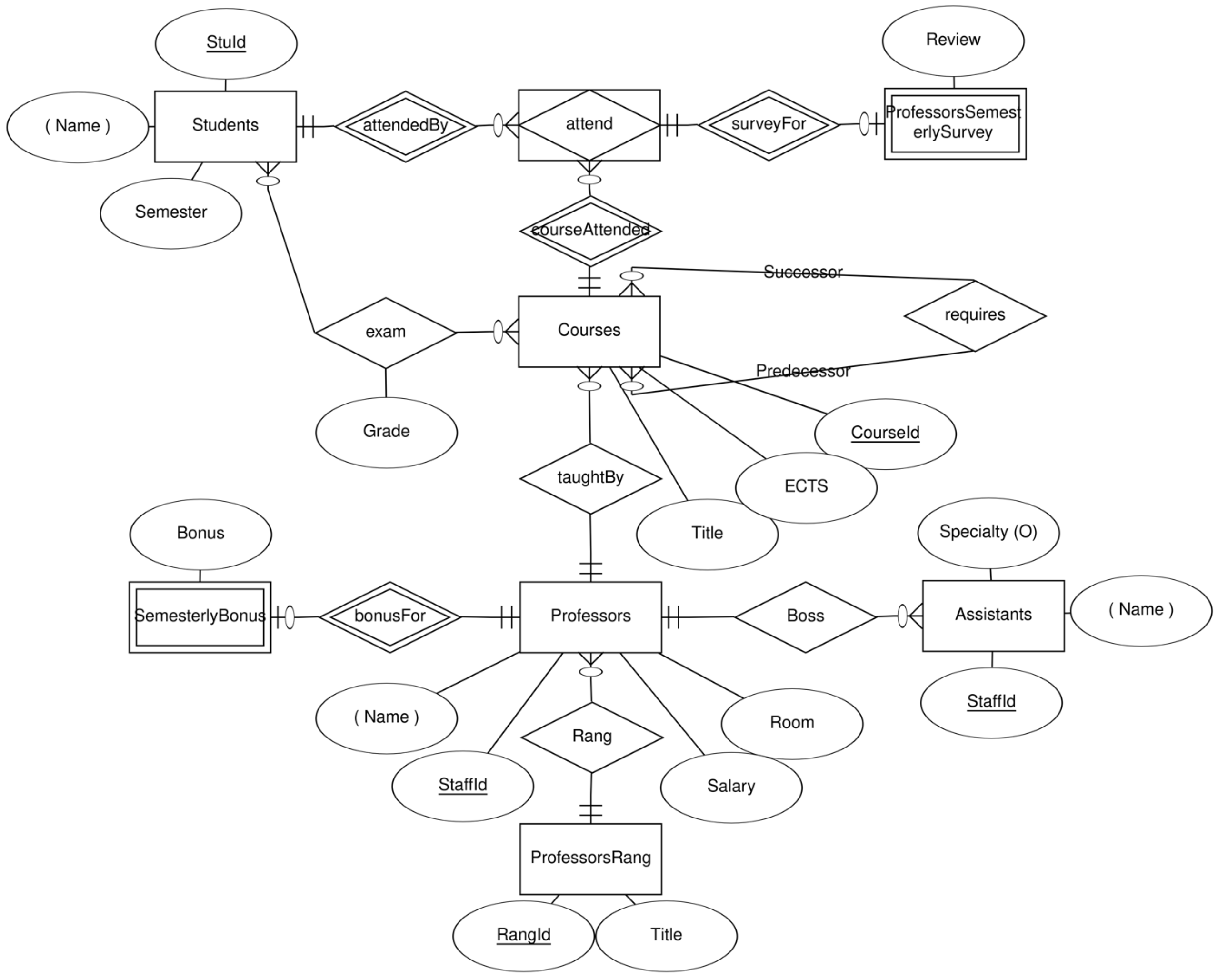
Example of portability: The system must be easy to deploy in different operating systems or database systems.
Many of the quality requirements, like usability, reliability and supportability are partially dependent and fulfilled by the database vendor. On our part though, we have to make sure to design the analysis model correctly so that requirements like usability and reliability get fulfilled.

Concerning pseudo-requirements (like legal requirements) we do not care much since it is intended to be a school project.

We move on to developing an entity relationship diagram to serve as our analysis model.

ANALYSIS MODEL

Entity Relationship Diagram of university database



Analysis of Entity Relationship Diagram

- Professors entity: has a StaffId as primary key, Name (which can be composite), Salary and a Room which is unique.
- ProfessorsRang entity: has a RangId as primary key and a Title.
- Rang relationship: one-to-many. Each professor is assigned a Rang from ProfessorsRang. The higher the RangId, the greater the importance of the professor.
- Assistants entity: has a StaffId as primary key, Name (which can be composite), and Specialty (which can be null).
- Boss relationship: one-to-many. Each assistant has a professor as his boss.
- SemesterlyBonus weak entity: has a Bonus attribute.
- bonusFor identifying relationship: one-to-one. Each SemesterlyBonus is intended for a professor.
- Courses entity: has a CourseId as primary key, ECTS and Title.
- requires relationship: many-to-many. Each course can be a predecessor to another course.
- taughtBy relationship: one-to-many. Each course is taught by a professor.
- Students entity: has a StuId as primary key, Name (which can be composite) and a Semester attribute.
- exam relationship: many-to-many. Has a Grade attribute. Each student can give exams for many courses (but only one exam for one course) and each course can have many students giving exams for that course.
- attend associative entity: primary key is comprised of the primary keys of Students and Courses.
- attendedBy identifying relationship: one-to-many. Each student can attend multiple courses.
- courseAttended identifying relationship: one-to-many. Each course can be attended by many students.
- ProfessorsSemesterlySurvey weak entity: has a Review attribute.
- surveyFor identifying relationship: each survey for a professor is based on a student's attendance to his course.

Solution Domain

SYSTEM DESIGN

There is not much work to do here, since we have set an architectural requirement to use MySQL as our database. User Access Definitions are supplied in a file called "user management.sql"

OBJECT DESIGN

Let us now convert this ER Diagram into a relational schema that will serve as a basis for implementing the DDL code of our tables.

Relational Schema of university database

Note: underlined denotes a primary key, cursive denotes a foreign key.

Professors {[StaffId : integer, Name : string, *RangId : integer*, Room : integer, Salary : integer]}
ProfessorsRang {[RangId : integer, Title : string]}
SemesterlyBonus{[*StaffId : integer*, Bonus : integer]}
Assistants{[StaffId : integer, Name : string, Specialty : string, *Boss : integer*]}
Courses {[CourseId : integer, Title : string, ECTS : string, *taughtBy : integer*]}
requires {[*Predecessor : integer, Successor : integer*]}
Students {[StuId : integer, Name : string, Semester : integer]}
exam {[*StuId : integer, CourseId : integer*, Grade : integer]}
attend {[*StuId: integer, CourseId: integer*]}
ProfessorsSemesterlySurvey {[*StuId : integer, CourseId : integer*, Review : integer]}

Other activities

We can now continue with implementation, testing (solution domain) and delivery (application domain). The resultant files are uploaded together with this report. Maintenance is out of the scope of this project.