FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

departamento de engenharia informática

BD Project

University Management System

João Ferreira, Guilherme Moreira & Rodrigo Manão

2023217920 | 2023222537 | 2023207589

Index

Introduction

- In the context of higher education, the efficient management of academic and administrative information is essential for the proper functioning of universities, making it crucial to adopt technological solutions that allow organizing, storing, and processing large volumes of data in a structured and secure manner. In this project, our goal is to develop a University Management System. This system handles the course administration, student enrollment, class scheduling, financial tracking, and evaluation processes.

Entity-Relation Model Description

- In the relationship between the entities *edition* and *class*, we considered *class* as a weak entity since classes only exist when associated with a specific course edition. We also reflected on the possibility of not treating it as a weak entity to accommodate scenarios such as standalone or open classes, but we choose not to explore that case further in this model.
- We consider *enrollment* a weak entity in relation to *student*, as enrollments only exist if there is a student to enroll, and a student can have multiple enrollments.
- *Evaluation* is weak in relation to *evaluation_period* because the evaluation belongs to a period, therefore, it can't be an evaluation without a period.
- *Evaluation_period* is weak in relation to edition because it belongs to an edition of a course and therefore there can't be an evaluation period without an edition. With this evaluation_period can be identified through its name and its corresponding edition.

- In this project, we consider that only students have a financial account, as it is not explicit in the statement that employees must have a financial account.
- We decided to create the *assistant_instructor* and *coordinator_instructor* entities to better understand the exercise. However, following the project statement, it would not be necessary, as they could be included in the instructor entity, manipulating this through relationships.
- *Payment* entity is used to activate triggers in *financial_acount* entity, meaning it will be responsible for adding/taking money from student accounts.
- The *grade_log* entity is used to save student grade logs among every degree program and is associated with a specific edition.
- During the ER elaboration, we thought of creating a University entity, which would be directly assigned to a department, where the classroom was located. However, we thought that it was not needed, so we kept the entity *building* only.
- We decided to relate the *staff* entity with student, person (maintaining heritage relation) and enrollment, so we can keep track of the staff who added that registries.
- In some entities, we decided to add the id attribute and it will become very important, in payments, person ids, class ids, etc… Principal entities in our diagram will have an id because of that. All of that to keep identifying these registers with a simple attribute.
- We created a *scheduled* entity, so a class can happen at various times a week, starting at a different time and having a different duration. Class type is just an attribute at *class* entity. The attendance is made in the physical model of the ER.
- Prerequisite courses are a loop relation, as the student needs other courses to be enrolled in some.
- Our database model follows the ACID principles (Atomicity, Consistency, Isolation, and Durability), maintaining the integrity and reliability of the stored data.
- In the physical model, the "distrito" attribute in the Person entity was not translated to "district". This does not affect the program and was

corrected in the ER model, but the physical model was not updated in time. The system works correctly and is not affected.

- In the future, the ER model might be adjusted if needed during the implementation. Any changes will be justified in a later report.

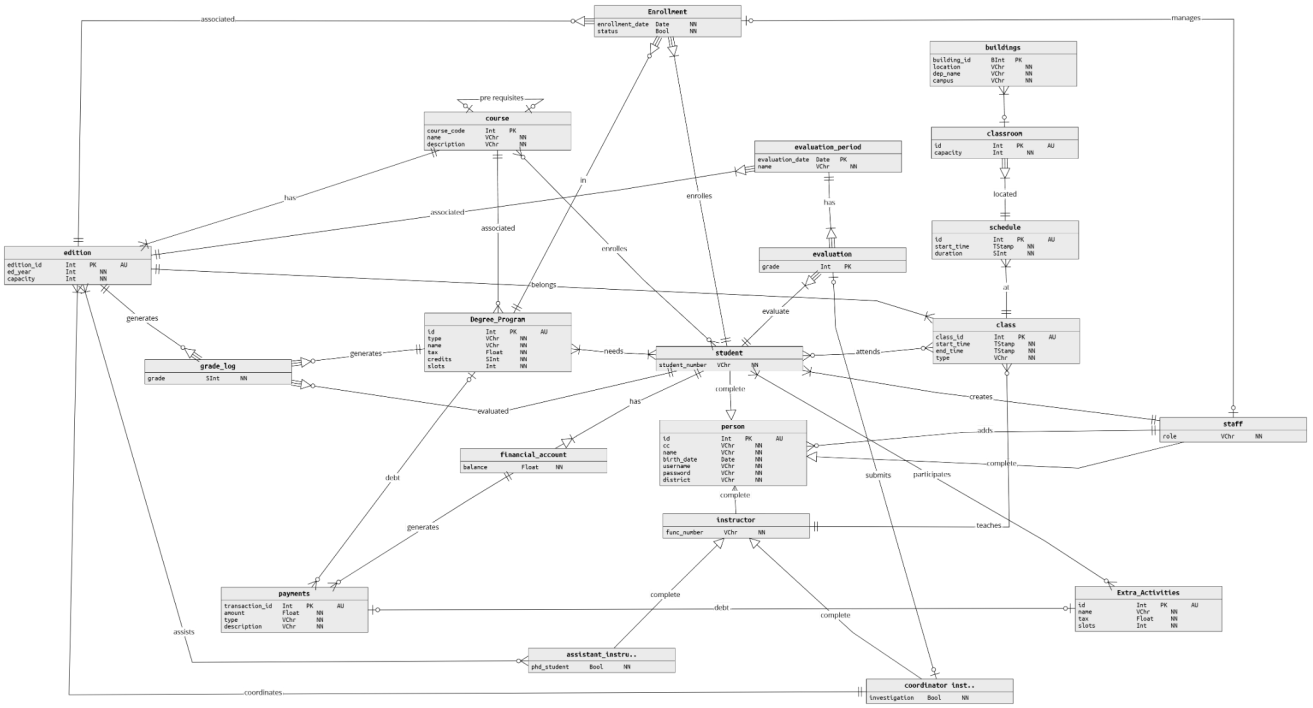

Fig. 1 - Entity-Relation Model

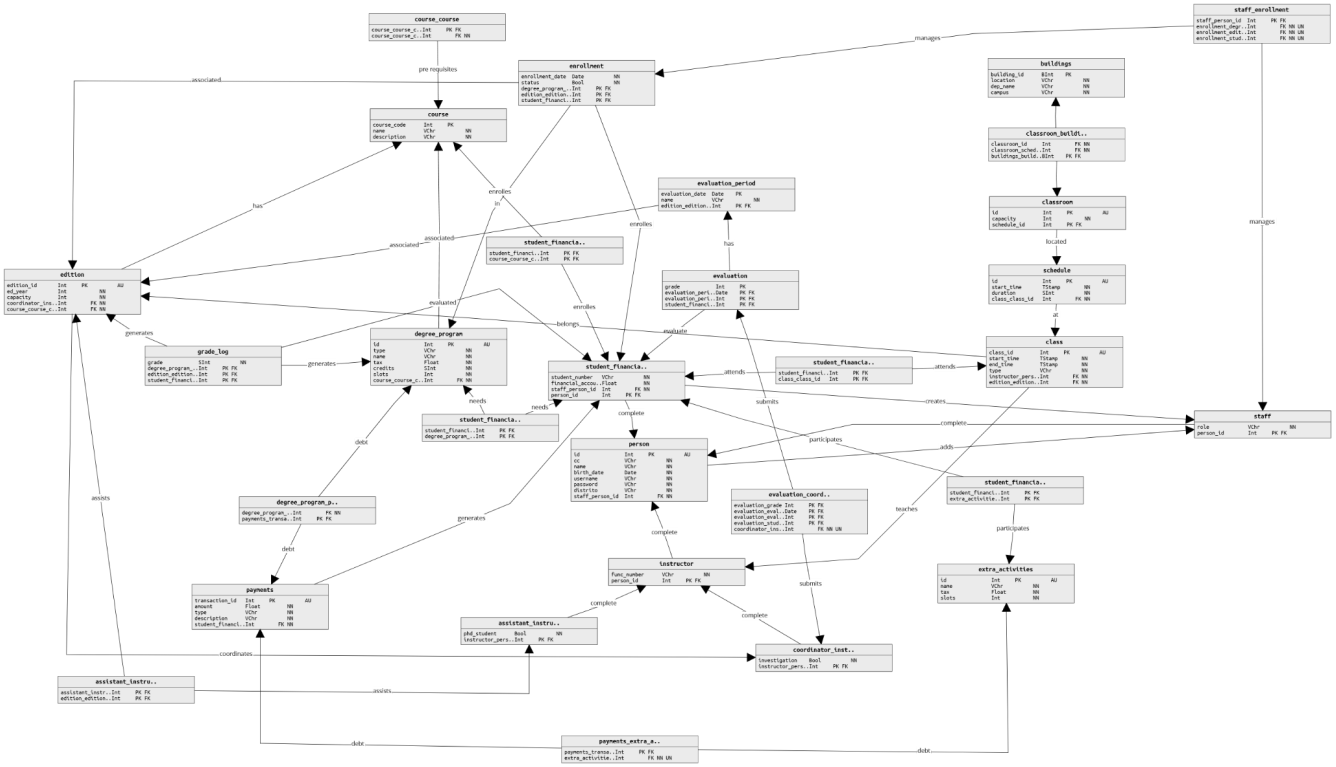

Fig. 2 - Relational Data Model

Transactions

- Transactions are blocks of operations that **must be executed entirely or rolled back in case of failure** to ensure the consistency of the database. In this case, some critical scenarios require transactions.

- ***Course enrollment***
    - In this transaction, a student enrolls in a course, requiring multiple operations to be completed successfully. First, the system verifies whether the student meets the course prerequisites and checks for available slots in the course edition. Second, it verifies if the classes he wishes to attend still have available slots and if the student has enough balance in his account. If everything is in order, the enrollment is recorded, the student's financial account and course edition capacity are updated, and a transaction entry is created. However, if anything goes wrong at any step, the entire process is rolled back, ensuring that no incomplete or incorrect data is saved.

- ***Activity registration***
    - In this transaction, a student registers in an activity and the associated fee is deducted from the student’s account. If the student withdraws from an activity, the charges will be recalculated. If anything goes wrong at any step, the entire process is rolled back, ensuring that no incomplete or incorrect data is saved.

- ***Grade Submission***
    - In this transaction, when a grade is submitted, the system first checks if the student is enrolled in the course, if so, the grade is submitted to the respective table together with its student and its

period. The academic averages of the approved students are also updated, within their degree programs.

Potencial concurrency conflicts

Concurrency conflicts occur when multiple transactions access the same data simultaneously, leading to inconsistencies or unexpected results, which have to be solved by the programmer itself. These conflicts can evolve deadlocks, long transactions,

- ***Financial account update***
  - If a student tries to pay a fee while his balance is being updated, there can be an incorrect calculation of the student's balance. This would be a problem because PostgresSQL doesn't block selects and default isolation level (Read Committed) does not prevent this issue. To avoid this, we can set an *access exclusive* lock or an *SELECT ... FOR UPDATE*, so that the balance can't be read until it is updated.

- ***Student enrollment***
  - If two students try to enroll in the same course edition at the same time, there is a risk of exceeding the maximum capacity. This happens because both transactions may check the available spots before committing, leading to over-enrollment. PostgreSQL's default isolation level, Read Committed, does not prevent this issue since it allows transactions to read outdated data.
  - To avoid this, we can use row-level locking with *SELECT ... FOR UPDATE*. This locks the course edition's row while checking availability, ensuring that only one student can enroll at a time. This

prevents race conditions while still allowing other operations to run smoothly.

- ***Activity register***
  - Here, the same concurrency problem as the student enrollment one could happen (two students trying to register at the same time), so to fix that, we can also use an exclusive lock, only allowing one update at a time.

Development Plan

Phases	Description	Task	Goals	João	Guilherme	Rodrigo	Status
Phase 1 (Week 1-3)	Planning & Database Design	Understanding Requirements	Read project documentation	☑	☑	☑	Done
			Identify key functionalities	☑	☑	☑	Done
		Design ER Diagram & Relational Model	Use ONDA for data modeling	☑	☑	☑	Done
			Define entities, attributes, relationships	☑	☑	☑	Done
		Define Transactions & Concurrency Strategie	Writing the introduction and observations	☑	☑	☑	Done
			Identify operations requiring transactions	☑	☑	☑	Done
			Define strategies for conflict handling	☑	☑	☑	Done
Phase 2 (Week 4-6)	Database Implementation & Basic API Setup	Implement Database Schema in PostgreSQL	Create tables, constraints, sequences	☐	☑	☐	Todo
			Write SQL scripts for database initialization	☐	☑	☐	Todo
		Develop REST API (Basic Functions)	Implement authentication	☑	☐	☑	Todo
			Set up Python backend	☑	☐	☑	Todo
			Create a simple retrieval endpoint	☑	☐	☑	Todo
Phase 3 (Week 7-10)	Core Features Implementation	Implement queries	User registration & authentication	☐	☑	☑	Todo
			Transaction operations implementations	☐	☑	☑	Todo
		Implement Business Logic & Triggers	Write pgSQL procedures and triggers	☑	☐	☐	Todo
			Ensure automatic updates for financial records, grades, etc.	☑	☐	☐	Todo
Phase 4 (Week 11-12)	Testing, Debugging & Finalizing	Complete REST API & Test Endpoints	Use Postman to validate API	☑	☑	☑	Todo
			Ensure security & transaction management	☑	☑	☑	Todo
		Final Documentation & Video Demo	Write installation & user manuals	☑	☑	☑	Todo
			Record a 5-minute demo video	☑	☑	☑	Todo

Fig. 3 - Table with development plan among all semester

Conclusion

In this midterm delivery, we were able to develop the basis of our project, which will then allow us to develop the proposed system.

With the Entity-Relation diagram and using the ONDA platform for its development, it was possible to observe the relationships and assign attributes to the respective entities.

In the remainder of the project, the RestAPI will be developed, together with the DBMS configuration and respective authentication functionalities.