Project Report

Data Storage Paradigms, IV1351

Edward Sharp, Leo Karsikas

20-11-2025

Project members:
Edward Sharp, edwardks@kth.se
Leo Karsikas, leoka@kth.se

GitHub repository link

Declaration:

By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.

It is furthermore declared that the solution below is a contribution by the project members only, and specifically that no part of the solution has been copied from any other source (except for lecture slides at the course IV1351), no part of the solution has been provided by someone not listed as a project member above, and no part of the solution has been generated by a system.

1 Introduction

The primary objective of this project was to perform a conceptual design of a database and later bridge it into a practical database implementation. The task involved translating a pre-defined, and rather sparse, conceptual model into a fully realised logical and physical database model, whilst ensuring the result is robust and normalised.

The project was divided into a mandatory and higher grade part, both of which are addressed in this report.

1.1 Mandatory requirements

The core requirements were to design a logical model using crow's foot notation that accurately represents the entities and relationships described in the project description.

The model had to adhere to the third normal form, or if not, have a good reason not to. The project also required the creation of a functional database using SQL.

1.2 Higher grade part

The database had to contain all necessary business rules and constants within the database itself, rather than relying on storing them in an external applications logic. This includes storing configurational data, such as the limit of how many courses a teacher may be a part of per period. The model also had to handle changes in data, for example, it needed to handle the fact that an employees salary could change over time and track it. Another example of this is the fact that the course layout of a course could change, for example, by adjusting the HP the course gives.

We worked on the project by ourselves, only consulting within the group.

2 Literature Study

The preparation for the **Task 1** began with going through all of the given material. Among them are lectures on normalisation and logical and physical models from Canvas for module 1 and also from the fifth to eighth chapters from *Fundamentals in Databases* book. The slides from the lab session were also helpful for this report.

The comprehended information was then later used during the creation of the logical and physical model, and also creation of the PostgreSQL Database. The recorded lecture with a step-by-step guide to the model was also reviewed.

3 Method

For this project, a couple of tools were used to implement the *Logical and Physical models* and the database. The logical and physical model was made using Astah Professional, and the database was made using PostgreSQL. The project group followed the setup steps provided in the software section of the Canvas page. In addition, a GitHub repository was created to further encourage collaboration in the project group and make the solution available to the examiner.

3.1 Logical and the Physical model

Since the whole task revolves around the creation of a Logical and Physical model, the recorded lecture was carefully followed together with the tips and tricks provided for the task.

The project group started with the exploration of the Conceptual Model provided for the task, and after that continued with adding necessary attributes, such as visibility for NOT NULL, key attributes, and types in the column row.

3.2 Primary and Foreign Keys

The key part of the Model is focusing on specifying the correct Primary keys and Foreign Keys between tables. To decide on the correct key for the table, discussion was encouraged within the group to provide the group with the opinions and insights from each group member.

Moreover, a focus on the cardinality of each relation was made during the creation of each new table, and later reviewed during the completion of the whole model.

3.3 Normalisation

The resulting model was continuously checked against the rules of the **Third Normal Form (3NF)**. The group specifically examined relationships to eliminate dependencies where a non-key attribute depends on another non-key attribute (transitive dependencies). Furthermore, to satisfy 1NFs requirement that every column in every row must contain only an atomic value, Complex many-to-many relationships were checked and, at times, resolved through the creation of explicit junction tables.

3.4 Implementation of Business Rules and Salary History

After the structural design, the group focused on integrating the specified business logic into the physical model. This involved creating specific tables to store numerical constants, such as the maximum number of courses an employee can teach in a given period. The constraint was enforced using a PSQL Trigger designed to check the data during insertion. The model was also adopted to handle changes over time by creating dedicated history tables and implementing versioning on the course_layout.

3.5 Creation of the database

Creation of the database was made after the first SQL draft export from Astah to setup the database.

The group decided to use a Python script to generate the required data for the database, since the provided tool for generating data was insufficient or too complex to use for the project, given the time constraints of the project.

After all of the steps mentioned above, the project group consulted with the assessment criteria document for this task to verify the validity of the proposed solution.

4 Result

The diagram presented in Figure 1 represents the proposed solution for both the Mandatory Part of the task and the Higher grade Part of the task. The ER diagram follows crown notation and contains attributes, entities, relations, and notes.

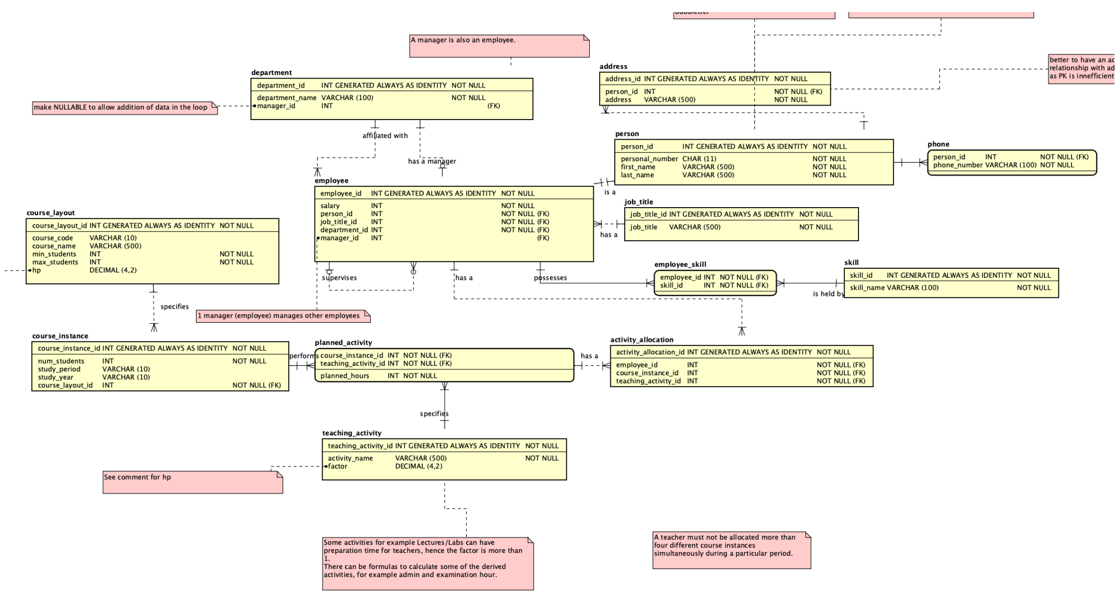

A manager is also an employee.
department
department_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
department_name VARCHAR (100) NOT NULL
manager_id INT (FK)
make NULLABLE to allow addition of data in the loop
affiliated with
has a manager
employee
employee_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
salary INT NOT NULL
person_id INT NOT NULL (FK)
job_title_id INT NOT NULL (FK)
department_id INT NOT NULL (FK)
manager_id INT (FK)
supervises
has a
possesses
1 manager (employee) manages other employees
course_layout
course_layout_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
course_code VARCHAR (10)
course_name VARCHAR (500)
min_students INT NOT NULL
max_students INT NOT NULL
hp DECIMAL (4,2)
specifies
course_instance
course_instance_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
num_students INT NOT NULL
study_period VARCHAR (10)
study_year VARCHAR (10)
course_layout_id INT NOT NULL (FK)
performs
planned_activity
course_instance_id INT NOT NULL (FK)
teaching_activity_id INT NOT NULL (FK)
planned_hours INT NOT NULL
has a
specifies
teaching_activity
teaching_activity_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
activity_name VARCHAR (500) NOT NULL
factor DECIMAL (4,2)
See comment for hp
Some activities for example Lectures/Labs can have preparation time for teachers, hence the factor is more than 1.
There can be formulas to calculate some of the derived activities, for example admin and examination hour.
address
address_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
person_id INT NOT NULL (FK)
address VARCHAR (500) NOT NULL
better to have an ac relationship with ad as PK is innefficient
person
person_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
personal_number CHAR (11) NOT NULL
first_name VARCHAR (500) NOT NULL
last_name VARCHAR (500) NOT NULL
is a
phone
person_id INT NOT NULL (FK)
phone_number VARCHAR (100) NOT NULL
job_title
job_title_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
job_title VARCHAR (500) NOT NULL
has a
employee_skill
employee_id INT NOT NULL (FK)
skill_id INT NOT NULL (FK)
skill
skill_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
skill_name VARCHAR (100) NOT NULL
is held by
activity_allocation
activity_allocation_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
employee_id INT NOT NULL (FK)
course_instance_id INT NOT NULL (FK)
teaching_activity_id INT NOT NULL (FK)
A teacher must not be allocated more than four different course instances simultaneously during a particular period.

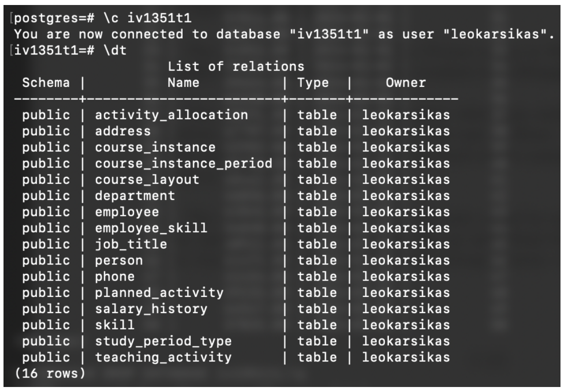
Figure 1: Logical and Physical Model

To ensure the uniqueness of primary keys, `INT GENERATED ALWAYS AS IDENTITY` was used for all dynamic, transactional tables (e.g., `employee`, `person`, and `course_instance`). This ensures a unique identifier is generated automatically for every new record. To preserve predictability, the `study_period_id` column was defined as a standard `INT` instead of utilizing `GENERATED ALWAYS AS IDENTITY`. This choice was made so that the ID can directly correlate to the period, (e.g., ID 1 always means period 1). If `study_period_id` was made to `INT GENERATED ALWAYS AS IDENTITY`, if a study period was accidentally deleted and later added back again, the ID for, for example, period 1, would change from 1 to something else.
Cardinality for each relationship has been defined according to its nature, and also considered if the relationship is identifying or non-identifying.

4.1 Database

The setup instructions for the database are described in **Instruction.md**.
Note: the instructions are made for Mac/Linux, and therefore cannot guarantee compatibility with Windows terminals.

The instructions file explains how to insert the setup script into the psql terminal, and also how to insert the generated data into the created database.
In the following figure and screenshot from the psql terminal is presented Figure 2



```
postgres=# \c iv1351t1
You are now connected to database "iv1351t1" as user "leokarsikas".
iv1351t1=# \dt
                  List of relations
 Schema |          Name          | Type  |    Owner
--------+------------------------+-------+-------------
 public | activity_allocation    | table | leokarsikas
 public | address                | table | leokarsikas
 public | course_instance        | table | leokarsikas
 public | course_instance_period | table | leokarsikas
 public | course_layout          | table | leokarsikas
 public | department             | table | leokarsikas
 public | employee               | table | leokarsikas
 public | employee_skill         | table | leokarsikas
 public | job_title              | table | leokarsikas
 public | person                 | table | leokarsikas
 public | phone                  | table | leokarsikas
 public | planned_activity       | table | leokarsikas
 public | salary_history         | table | leokarsikas
 public | skill                  | table | leokarsikas
 public | study_period_type      | table | leokarsikas
 public | teaching_activity      | table | leokarsikas
(16 rows)
```

Figure 2: Screenshot from the psql terminal

5 Discussion

The Logical and Physical model was based on the project description provided for the task, and thoroughly reviewed during the later stages of the project, to avoid missing details mentioned in the project description and in the assessment criteria. Among them, following the crown convention given in the Conceptual Model template, and also the self-explanatory naming of both table and attribute names.

For all of the columns in the model, primary and foreign keys are specified, with consideration of uniqueness. This is necessary for generating an ID for the majority of the primary keys. Moreover, the types of columns in the model were modified with regard to compatibility with the PostgreSQL database, and therefore, the use of the previously mentioned *INT ALWAYS GENERATED AS IDENTITY* was suitable for the implementation.

In regard to the relationship between the tables, do all of them have a specified cardinality and identifying or non-identifying relationship. Moreover, the majority of the relations have a name or an "action" that describes what the relations do in combination with the entity it belongs to.

5.1 Choice of Data Types

Choosing between data types for an attribute was sometimes more obvious than other times. For most values containing strings of variable length, for example, a persons name, course codes, and course names, `VARCHAR(n)` was used. The maximum length, n, was altered depending on the situation, for example, a course code is usually much shorter than the course name. In our case, these were set to `VARCHAR(10)` for course code, and `VARCHAR(500)` for course name. For `personal_number`, `CHAR(11` was used instead to ensure a uniform format for the personal numbers. The opted format for personal numbers was yymmdd-xxxx, often referred to as a 10-length personal number, but because of the dash (-) the length becomes 11.

Other data types worth mentioning are the use of `DECIMAL`. The factor, HP of a course, and salary_amount all used `DECIMAL` as their data types. This choice was made because these values are not integers, e.g., they can have decimal values, and `FLOAT` or `REAL` was avoided because of floating point errors. For example, `FLOAT` can suffer from binary rounding issues, where $0.1 + 0.2 \neq 0.3$. By using for example `DECIMAL (4, 2` for factor, there will be 4 total positions of value, where 2 of them are beyond the decimal place, for example 3.20 or at max 99.99.

5.2 History Preservation and Versioning

To handle changing course layouts, we designed the `course_layout` table to accept multiple records for the same `course_code`, distinguishing them by including a `valid_from` date. Similarly, the `salary` column was removed from the `employee` table and replaced with the dedicated `salary_history` entity. These changes ensure that when a course's HP or a teacher's salary is updated, a new record is created rather than overwriting the old one.

The main advantage to keeping multiple versions of data is that it guarantees data accuracy for all historical transactions. Since attributes like salary or course HP are tied to specific effective dates, creating a new record instead of overwriting the old one ensures that past records, for example a student's 2022 transcript or a 2023 payroll calculation, remain correct. Furthermore, it completely removes the risk of update anomalies, as a change in one version of a course definition cannot accidentally corrupt past records.

There are however disadvantages to this implementation. Firstly, it increases the complexity of queries. A developer can no longer simply select the employee's salary from the main table. Instead, they must use filtering, for example selecting the record with the maximum `valid_from date` to find the current rate. Secondly, since every minor change in a course layout or an employee's raise requires a new, complete record, the database will naturally grow faster.

5.3 Logic and Business Rules

A core requirement was that the database must manage all necessary data and logic, leaving no data to be stored in for example an application. This involves two major points: the storage of the constant value '4' and the enforcement of the rule.

The constraint that "a teacher can teach up to 4 courses in a particular period" was implemented by storing the value 4 in the `system_rules` table. This achieves the goal of keeping all crucial data within the database itself. The enforcement of the course limit is handled by the `check_allocation_limit` PSQL trigger, which is attached to the `activity_allocation` table. This ensures the business rule is enforced at the data layer. The trigger logic specifically:

- Retrieves the dynamic max limit from `system_rules`
- Counts the distinct `course_instance_ids` associated with the employee in the current `study_period_id`
- Aborts the transaction and raises an exception if the limit is exceeded.

This implementation ensures that the business requirement is met by the physical model.

5.4 Normalisation violation

The overall model is designed to achieve Third Normal Form (3NF), but a deliberate trade-off was made in the design of the `address` entity, resulting in a minor, but deliberate, violation of 3NF.

The `address` table stores the full street address. If we were to strictly adhere to 3NF, the following transitive dependency would need to be resolved:

$$\text{Address ID} \rightarrow \text{Postal Code} \rightarrow \text{City}$$

In a pure 3NF model, the city name is dependent on the postal code, not the primary key (`address_id`). Therefore, the city should be moved to a separate `postal_codes` lookup table.

The decision to keep the `city` directly within the `address` table is mainly due to simplicity in data entry. Creating and maintaining a full and comprehensive postal code database, that ideally covers address formats of different countries, introduces significant complexity that outweighs the small benefit of eliminating text redundancy in this application.

6 Comments About the Course

We approximately spent a total of 4 sessions of 4–5 hours each, where we met up and worked on the seminar together.