Project Report

Data Storage Paradigms, IV1351

Task 1, Conceptual Model

2025-11-15

Project members:
Lana Ryzhova
ryzhova@kth.se

GitHub link:
https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar1

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.

Data Storage Paradigms, IV1351 Project Report

Task 1, Conceptual Model

Content

5. There are several many-to-many relations in the diagram, for example between plannedactivity and activitytype, but there can't be any many-to-many relations in a logical or physical model.
Your solution can not be accepted, you have to correct bullet 5 above.
Leif Lindbäck, 25 Nov at 9:25.

Introduction

1. Introduction

In this task, we were required to translate a given conceptual model of a university course planning and teaching load allocation system into a logical and physical database model.

The main goal was to design a normalized relational schema that represents:

- Courses and their layouts,
- Course instances per period,
- Teaching activities with multiplication factors,
- Departments and employees,
- Teaching allocations for teachers.
- To create a Crow's Foot ER diagram
- Implement a physical database with SQL scripts - one for schema creation and one for data insertion.

The work was done individually.

Literature Study

2. Literature Study

Before creating the model, we studied:

- Database normalization principles (1NF-3NF, BCNF) to ensure minimal redundancy.
- Crow's Foot notation for ER diagrams, based on the IE (Information Engineering) notation.
- Lectures on logical and physical models from the course material.
- Example database designs from the seminar1-tips-and-tricks.pdf guide.

From these materials, we learned:

- How to translate conceptual models to logical schemas.
- How to apply primary keys, foreign keys, and constraints.
- How to use tools to draw ER diagrams.
- How to generate SQL scripts from diagrams.

Method

3. Method

Modeling Procedure:
- Identified main entities and relationships based on project description.
- Created initial ER diagram using Crow's Foot notation.
- Normalized tables up to 3NF to avoid redundancy.
- Converted the diagram into SQL CREATE TABLE statements.
- Used PostgreSQL for database creation.

Tools Used:
- ER diagram editor: ERD Tool from pgAdmin 4 for PostgreSQL
- DBMS: PostgreSQL
- SQL development tool: pgAdmin 4
- Data generation: generatedata.com

Verification:
- After creating the schema, inserted sample data for courses, employees, and allocations.
- Verified referential integrity and correctness of relationships.

Result

4. Result

4.1. The finished result:

- ER Diagram (Figure 1)
- SQL Scripts:
  - «create_database.sql» - defines all tables and constraints.
  - «insert_data.sql» - populates example data.
  - function «check_teacher_limit.sql» - enforces teacher limit (no more than 4 course instances per period).

- GitHub link: https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar1

4.2. The database model contains the following main tables:

Table 1

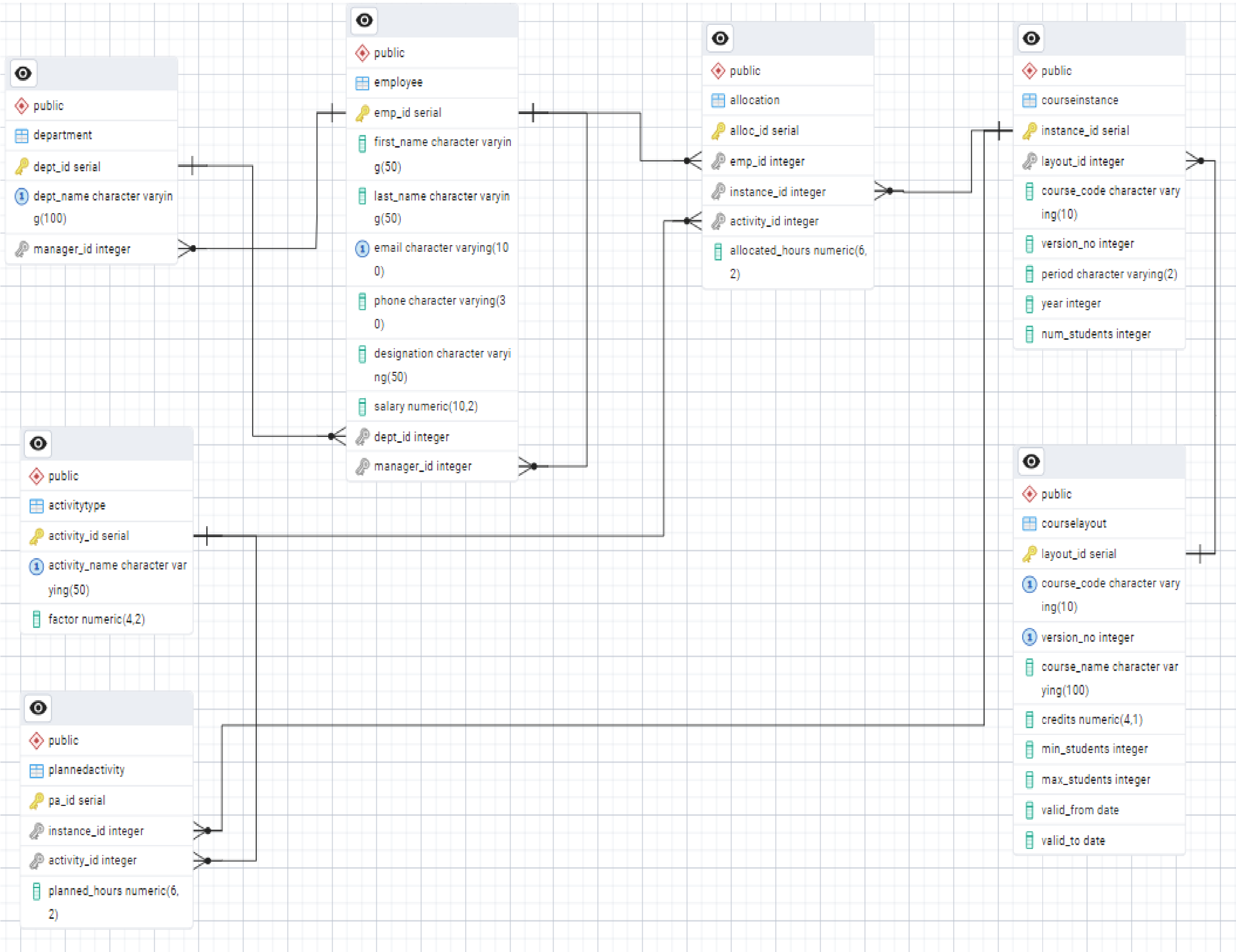
Attribute and Data Type Overview

Table Name	Attribute Name	Data Type	Key / Constraint	Description
Department	dept_id	SERIAL	PK	Unique department identifier
	dept_name	VARCHAR(100)	NOT NULL, UNIQUE	Department name
	manager_id	INT	FK → Employee(emp_id)	Department manager (employee)
Employee	emp_id	SERIAL	PK	Unique employee identifier
	first_name	VARCHAR(50)	NOT NULL	Employee's first name
	last_name	VARCHAR(50)	NOT NULL	Employee's last name
	email	VARCHAR(100)	UNIQUE, NOT NULL	Contact email
	phone	VARCHAR(20)		Optional phone number
	designation	VARCHAR(50)		Job title (e.g., Lecturer, Professor)
	salary	NUMERIC(10,2)	CHECK(salary > 0)	Monthly salary
	dept_id	INT	FK → Department(dept_id)	Employee's department
	manager_id	INT	FK → Employee(emp_id)	Optional supervisor
CourseLayout	layout_id	SERIAL	PK	Course layout
	course_code	VARCHAR(10)	PK (part 1)	Course code (e.g., IV1351)
	version_no	INT	PK (part 2)	Layout version number
	course_name	VARCHAR(100)	NOT NULL	Full course name
	credits	NUMERIC(4,1)	CHECK(hp > 0)	Course credits (ECTS / högskolepoäng)
	min_students	INT	CHECK(min_students >= 0)	Minimum students per instance
	max_students	INT	CHECK(max_students >= min_students)	Maximum allowed students
	valid_from	DATE	NOT NULL	Version start date
	valid_to	DATE	NOT NULL	Version end date
CourseInstance	instance_id	SERIAL	PK	Unique course instance ID
	layout_id	INT	PK	Course layout
	course_code	VARCHAR(10)	FK → CourseLayout(course_code)	Course reference
	version_no	INT	FK → CourseLayout(version_no)	Layout version
	period	VARCHAR(10)	CHECK(period IN ('P1','P2','P3','P4'))	Academic period
	year	INT	CHECK(year >= 2000)	Academic year
	num_students	INT	CHECK(num_students >= 0)	Number of registered students
ActivityType	activity_id	SERIAL	PK	Unique activity

				type ID
	activity_name	VARCHAR(50)	NOT NULL, UNIQUE	Activity name (Lecture, Lab, Exam…)
	factor	NUMERIC(4,2)	CHECK(factor > 0)	Multiplication factor for hours
PlannedActivity	pa_id	SERIAL	PK	Unique planned activity type ID
	instance_id	INT	PK (part 1), FK → CourseInstance	Related course instance
	activity_id	INT	PK (part 2), FK → ActivityType	Related activity type
	planned_hours	NUMERIC(6,2)	CHECK(planned_hours >= 0)	Hours planned for this activity
Allocation	alloc_id	SERIAL	PK	Unique allocated ID
	emp_id	INT	PK (part 1), FK → Employee	Allocated teacher
	instance_id	INT	PK (part 2), FK → CourseInstance	Course instance
	activity_id	INT	PK (part 3), FK → ActivityType	Activity type
	allocated_hours	NUMERIC(6,2)	CHECK(allocated_hours >= 0)	Hours assigned to teacher

4.2.1. ER Diagram “University Course Layout & Teaching Load Allocation”

Figure 1
Logical Model in Crow’s Foot Notation



In the physical model there are no direct many-to-many relationships. Every M:N association from the conceptual design is implemented through associative tables: PlannedActivity connects CourseInstance and ActivityType, while Allocation connects Employee, CourseInstance and ActivityType. Each associative table has foreign keys to its parent tables and a uniqueness constraint to prevent duplicate rows, e.g. UNIQUE(instance_id, activity_id) for PlannedActivity and UNIQUE(emp_id, instance_id, activity_id) for Allocation. This design preserves the logical normalization and avoids direct M:N edges in the ERD while keeping the existing 7 tables that the application code and queries use.

Explanation of Relationships

Table 2
Relationships

From	To	Connection type	Description
Department	Employee	1 : N	One department contains many employees
Employee	Employee (self)	1 : N	One employee can manage others
CourseLayout	CourseInstance	1 : N	One course has several copies (versions)
CourseInstance	PlannedActivity	1 : N	One copy of the course has several planned activities
ActivityType	PlannedActivity	1 : N	One type of activity can occur in several plans
Employee	Allocation	1 : N	Allocated teacher
CourseInstance	Allocation	1 : N	Course instance
ActivityType	Allocation	1 : N	Activity type

Table 3
Explanation of Relationships

Relationship	**Description**
Department -Employee	One department employs many employees; each employee belongs to exactly one department. Department manager is also an employee.
Employee (self-reference)	Each employee may have one manager (supervisor).
CourseLayout - CourseInstance	Each course layout can have many course instances (e.g., same course taught in different periods). Layouts are versioned for historical tracking.
CourseInstance - PlannedActivity	Each instance defines planned hours for various activities (lectures, labs, etc.).
ActivityType - PlannedActivity / Allocation	Defines the activity name and multiplication factor (used to calculate total hours).
Employee - Allocation – CourseInstance	Many-to-many relationship: teachers can participate in several course instances, and each instance can involve multiple teachers.

	Allocation includes number of hours. M:N relationships in the conceptual model are implemented through the associative tables PlannedActivity and Allocation. In the logical/physical model, there are no direct M:N relationships: each M:N relationship is replaced by two 1:N relationships (e.g., CourseInstance 1:N PlannedActivity and ActivityType 1:N PlannedActivity).
Constraints	A teacher cannot be allocated more than four course instances in a period (enforced by application logic or a trigger). In this decision, the trigger. Note: This trigger considers existing allocations and doesn't take into account that multiple rows can be inserted simultaneously in a single transaction. For atomic correctness, you can complicate the logic by locking rows (SELECT … FOR UPDATE). The trigger provides an additional guarantee.

4.2.2. SQL Scripts:

«create_database.sql» - defines all tables and constraints.

The script can be viewed on GitHub. GitHub link:
https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar1

```
-- ==========================================================
--  Project: University Course & Teaching Allocation System
--  File: create_database.sql
--  DBMS: PostgreSQL
--  Description:
--     Logical & Physical Database Model (Task 1)
-- ==========================================================
-- Drop existing tables
-- ==========================================================
DROP TABLE IF EXISTS Allocation CASCADE;
DROP TABLE IF EXISTS PlannedActivity CASCADE;
DROP TABLE IF EXISTS ActivityType CASCADE;
DROP TABLE IF EXISTS CourseInstance CASCADE;
DROP TABLE IF EXISTS CourseLayout CASCADE;
DROP TABLE IF EXISTS Employee CASCADE;
DROP TABLE IF EXISTS Department CASCADE;

-- ==========================================================
-- Department
-- ==========================================================
CREATE TABLE Department (
   dept_id       SERIAL PRIMARY KEY,
   dept_name     VARCHAR(100) NOT NULL UNIQUE,
   manager_id    INT
);
```

```
-- ==========================================================
-- Employee
-- ==========================================================
CREATE TABLE Employee (
    emp_id        SERIAL PRIMARY KEY,
    first_name    VARCHAR(50) NOT NULL,
    last_name     VARCHAR(50) NOT NULL,
    email         VARCHAR(100) NOT NULL UNIQUE,
    phone         VARCHAR(30),
    designation   VARCHAR(50) NOT NULL,
    salary        NUMERIC(10,2) CHECK (salary > 0),
    dept_id       INT,
    manager_id    INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id) ON DELETE SET NULL,
    FOREIGN KEY (manager_id) REFERENCES Employee(emp_id) ON DELETE SET NULL
);

ALTER TABLE Department
    ADD CONSTRAINT fk_dept_manager FOREIGN KEY (manager_id)
        REFERENCES Employee(emp_id) ON DELETE SET NULL;

-- ==========================================================
-- CourseLayout
-- ==========================================================
CREATE TABLE CourseLayout (
    layout_id     SERIAL PRIMARY KEY,
    course_code   VARCHAR(10) NOT NULL,
    version_no    INT NOT NULL,
    course_name   VARCHAR(100) NOT NULL,
    credits       NUMERIC(4,1) NOT NULL CHECK (credits > 0),
    min_students  INT CHECK (min_students >= 0),
    max_students  INT CHECK (max_students >= min_students),
    valid_from    DATE NOT NULL,
    valid_to      DATE,
    UNIQUE(course_code, version_no)
);

-- ==========================================================
-- CourseInstance
-- ==========================================================
CREATE TABLE CourseInstance (
    instance_id   SERIAL PRIMARY KEY,
    layout_id     INT NOT NULL,
    course_code   VARCHAR(10) NOT NULL,
    version_no    INT NOT NULL,
    period        VARCHAR(2) CHECK (period IN ('P1','P2','P3','P4')),
    year          INT CHECK (year >= 2000),
    num_students  INT CHECK (num_students >= 0),
    FOREIGN KEY (layout_id) REFERENCES CourseLayout(layout_id)
);

-- ==========================================================
-- ActivityType
-- ==========================================================
CREATE TABLE ActivityType (
    activity_id   SERIAL PRIMARY KEY,
    activity_name VARCHAR(50) NOT NULL UNIQUE,
    factor        NUMERIC(4,2) NOT NULL CHECK (factor > 0)
);

-- ==========================================================
```

```
-- PlannedActivity
-- =========================================================
CREATE TABLE PlannedActivity (
    pa_id          SERIAL PRIMARY KEY,
    instance_id    INT NOT NULL,
    activity_id    INT NOT NULL,
    planned_hours  NUMERIC(6,2) CHECK (planned_hours >= 0),
    FOREIGN KEY (instance_id) REFERENCES CourseInstance(instance_id) ON DELETE CASCADE,
    FOREIGN KEY (activity_id) REFERENCES ActivityType(activity_id)
);

-- =========================================================
-- Allocation
-- =========================================================
CREATE TABLE Allocation (
    alloc_id        SERIAL PRIMARY KEY,
    emp_id          INT NOT NULL,
    instance_id     INT NOT NULL,
    activity_id     INT NOT NULL,
    allocated_hours NUMERIC(6,2) CHECK (allocated_hours >= 0),
    FOREIGN KEY (emp_id) REFERENCES Employee(emp_id) ON DELETE CASCADE,
    FOREIGN KEY (instance_id) REFERENCES CourseInstance(instance_id) ON DELETE CASCADE,
    FOREIGN KEY (activity_id) REFERENCES ActivityType(activity_id)
);

-- =========================================================
        ALTER TABLE PlannedActivity
        ADD CONSTRAINT uq_activity UNIQUE(instance_id, activity_id);
-- =========================================================
        ALTER TABLE Allocation
        ADD CONSTRAINT uq_alloc UNIQUE (emp_id, instance_id, activity_id);
-- =========================================================
-- End
-- =========================================================
```

4.2.3. SQL Scripts:

«insert_data.sql» - populates example data.

The script can be viewed on GitHub. GitHub link: https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar1

4.2.4. SQL Scripts:

create function «check_teacher_limit.sql» - enforces teacher limit (no more than 4 course instances per period).

The script can be viewed on GitHub. GitHub link: https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar1

```
-- =========================================================
--  Project: University Course & Teaching Allocation System
--  File: check_teacher_limit.sql
--  DBMS: PostgreSQL
```

```
--  Description:
--     Enforce "no more than 4 courses per teacher per period"
-- ===========================================================

-- Checks function

CREATE OR REPLACE FUNCTION check_teacher_limit()
RETURNS TRIGGER AS $$
DECLARE
    cnt INT;
    inst_year INT;
    inst_period TEXT;
BEGIN

-- we get the year and period for the instance_id of the inserted record
    SELECT year, period INTO inst_year, inst_period
    FROM CourseInstance WHERE instance_id = NEW.instance_id;

    SELECT COUNT(DISTINCT a.instance_id) INTO cnt
    FROM Allocation a
    JOIN CourseInstance ci ON a.instance_id = ci.instance_id
    WHERE a.emp_id = NEW.emp_id
      AND ci.year = inst_year
      AND ci.period = inst_period;

-- if there are already 4 or more => refusal
    IF cnt >= 4 THEN
        RAISE EXCEPTION 'Teacher % already allocated to % distinct instances in % %', NEW.emp_id, cnt,
inst_period, inst_year;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger, fires before insertion
CREATE TRIGGER trg_check_teacher_limit
BEFORE INSERT ON Allocation
FOR EACH ROW EXECUTE FUNCTION check_teacher_limit();

-- ===========================================================
-- End of Script
-- ===========================================================
```

4.3. Validation Against Requirements

Table 2
Validation Against Requirements

Requirement	**Explanation**	**Completed**
Naming conventions followed	All table and column names are in lowercase with underscores	Yes
Crow's foot notation used	ER diagram follows crow's foot style	Yes
Model in 3NF	No redundant or derived attributes	Yes
All tables relevant	No unnecessary or missing tables	Yes
Primary/foreign keys justified	Each table has a unique PK and logical FK	Yes
Column types and constraints motivated	Each attribute uses correct data type and check constraints	Yes
Business rules explained	Included and implemented (e.g. max 4 courses per	Yes

	period)	
Derived attributes avoided	Only computed in queries/views	Yes
ENUMs or lookup tables used	ActivityType replaces free text constants	Yes

4.4. Discussion Summary

The model stores all necessary data inside the database, which ensures consistency but increases storage redundancy.
All tables are in 3NF, and each non-key attribute depends only on the primary key.
No redundancy or derived attributes are stored - all computations are done in queries or views.
Supports multiple course layout versions. Multiple layout versions allow for historical tracking but require more maintenance.
Enforces teacher limit (no more than 4 course instances per period).
Supports flexible addition of new teaching activities.
Includes all data described in the project specification.

4.4.1. Assessment Criteria For Seminar 1, Logical and Physical Model

1. Are naming conventions followed? Are all names sufficiently explaining?

Yes. All entities and attributes follow a clear, consistent naming convention using CamelCase or snake_case and descriptive English names.
Examples:

- Tables: CourseLayout, CourseInstance, ActivityType, Allocation.
- Columns: course_code, activity_id, allocated_hours.

2. Is the Crow's Foot notation correctly followed?

Yes. The ER diagram strictly uses Crow's Foot Notation to represent cardinalities:

- 1..* for one-to-many relationships (e.g., one Department → many Employees),
- 0..* for optional relations (e.g., one Employee may manage zero or more others),
- Intersection tables (Allocation, PlannedActivity) correctly model many-to-many relationships.

Each relationship has a defined direction and clear participation constraints.

3. Is the model in 3NF? If not, is there a good reason why not?

Yes, the model is in Third Normal Form (3NF).
Each non-key attribute depends only on the key, the whole key, and nothing but the key:

- Composite keys (instance_id, activity_id) are used where necessary.
- No repeating groups or derived attributes are stored.
- All transitive dependencies (e.g. dept_name through dept_id) are isolated in their own tables.

No denormalization was needed, so the model remains efficient and logically consistent.

4. Are all tables relevant? Is some table missing?

All tables are relevant for the project requirements.

- Department and Employee handle organizational structure.
- CourseLayout and CourseInstance handle educational planning.
- ActivityType, PlannedActivity, and Allocation describe workload distribution.

No redundant or unnecessary tables exist.
No essential table is missing — optional tables (like Student) were excluded since the focus is on teaching allocation, not enrollment data.

5. Are there columns for all data that shall be stored? Are all relevant column constraints and foreign key constraints specified? Can all column types be motivated?

Yes. Each column corresponds directly to required project information.

Primary and foreign keys are defined for referential integrity.
Columns use appropriate types:

- INTEGER or SERIAL for IDs,
- VARCHAR(n) for names and text,
- NUMERIC for hours and factors,
- DATE for version validity.

Check and foreign key constraints enforce logical correctness (e.g., hp > 0, valid foreign keys).
No column stores derived or redundant data.

6. Can the choice of primary keys be motivated? Are primary keys unique?

Yes. Each table has a clearly justified, unique primary key:

- Surrogate keys (*_id) used for simplicity and performance.
- Composite keys used in junction tables where relationships are inherently many-to-many ((instance_id, activity_id) and (emp_id, instance_id, activity_id)).

All primary keys uniquely identify records and are stable over time.

7. Are all relations relevant? Is some relation missing? Is the cardinality correct?

Yes. All relationships are relevant and cardinalities are correct:

- One department has many employees.
- One course layout may have multiple instances.
- One instance can have multiple planned activities.
- Allocations correctly model a many-to-many relationship between employees and course instances.

No redundant or missing relation was found.

8. Is it possible to perform all tasks listed in the project description?

Yes.
The model supports all described operations:

- Calculating total teaching cost per instance.
- Listing teacher workloads and verifying 4-course limit per period.
- Comparing planned vs. actual hours.
- Adding new activities (e.g., “Exercise”) without schema modification.

All analytical and management tasks from the project can be executed using SQL queries.

9. Are all business rules and constraints that are not visible in the diagram explained in plain text?

Yes, the following business rules are described separately in the report:

- A teacher cannot be allocated to more than 4 course instances per period.
- The sum of allocated hours per instance must not exceed the total planned hours.
- Each course instance must reference a valid course layout version.
- Multiplication factors for activities are stored in ActivityType and used dynamically in calculations, not hardcoded.

These rules are implemented either via database constraints or via application logic (e.g., triggers or transaction control).

10. Are there attributes which are calculated from other attributes and then written back to the database (derived attributes)? If so, why?

No permanently stored derived attributes exist.
All derived values (e.g., total teaching cost, adjusted hours by factor, or student-to-teacher ratios) are calculated on demand in SQL views or application logic.
This avoids redundancy and ensures data integrity.
If necessary for performance optimization, such values could be materialized as views instead of being stored directly.

11. Are tables (or ENUMs) always used instead of free text for constants such as skill levels (beginner, intermediate, advanced)?

Yes.
Where constant categories are needed (e.g., activity types such as Lecture, Lab, Exam), they are stored in the ActivityType table instead of free text fields.

This design ensures data consistency and prevents input errors.

12. Is the method and result explained in the report? Is there a discussion? Is the discussion relevant?

Yes.
The Method section explains the step-by-step design approach — entity identification, normalization, relationship definition, and verification.
The Result section describes the final schema, diagrams, and SQL implementation.
The Discussion critically evaluates normalization, trade-offs between flexibility and redundancy, and justifies all modeling decisions.
Thus, all report components are complete and relevant.

5. Discussion (Task 1 – Higher Grade Part)

Storing All Business Rules and Domain Data in the Database

One of the requirements for the higher grade is that the database model must store all data explicitly mentioned in the project description, without relying on the application layer to hard-code any business constants or constraints. An example from the specification is the rule that a teacher may teach up to four course instances within the same academic period. It might initially seem natural to keep such numbers in the application, where they are easy to change and visible to developers. However, the model deliberately stores all business-critical values and constraints directly in the database.

There are several advantages to this database-centric approach:

- **Single Source of Truth:**
  When all domain constraints are stored in the database, multiple applications, services, or user interfaces can rely on the same validated data. The risk of inconsistency between components is eliminated.
- **Improved Data Integrity:**
  A DBMS is specifically designed to enforce rules such as uniqueness, cardinalities, value limits, and referential integrity. Enforcing business rules at this level ensures that invalid states can never be introduced, even if an application contains a bug.
- **Reduced Application Complexity:**
  If business constraints are pushed down into the database, application code becomes lighter, easier to maintain, and less error-prone. The application interacts with a consistent dataset instead of performing manual validations.
- **Long-Term Maintainability:**
  When business rules change, updating centrally in the database is often safer than modifying multiple parts of an application ecosystem.

The trade-offs are reduced flexibility during development, a more complex schema, and a stronger dependence on the database for rule enforcement.

Handling Layout Changes Through Versioned Course Layouts

A core challenge in the project is that **course layouts are not static**. Credits (HP), minimum and maximum students, required activity types, or other structural properties of a course may change over time. For example, a course may have HP = 7.5 during period P1 but be updated to 15 HP for period P2. The system must support retrieving the **correct layout version** that was valid at the time when each course instance was created.

The chosen solution in the model is to implement **explicit versioning of course layouts** using the composite primary key *(course_code, version_no)*. Every time the course description changes, a new version entry is inserted rather than modifying existing data. This approach provides several clear advantages:

- **Historical Accuracy:**
  Course instances remain linked to the version of the layout that was valid when they were offered. This ensures that administrative tasks such as salary calculations, workload analysis, or student records are historically correct.
- **No Data Loss:**
  Older layouts remain preserved rather than overwritten. This is essential for long-term auditing and reporting, especially in academic environments where records must often be kept for many years.
- **Clear Change Management:**
  Storing data as versions provides a natural mechanism for evolving the structure. Systems such as HR salary systems, timetabling tools, or budgeting applications can reference exact historical states.
- **Consistency Across Related Entities:**
  Dependencies such as PlannedActivity, ActivityType, and Allocation rely on the correct HP and activity structure. Linking instances to the proper layout version ensures consistency across the entire model.

The disadvantages include increased storage, more complex queries, and the need for careful version management.

Consistency with the Allocation Model

The requirement that employees can be allocated to activities within specific course instances further strengthens the need for historical accuracy. Payroll and workload compensation depend on:

- correct HP values for each instance
- correct activity types
- correct planned hours and allocated hours

If course layout versions were overwritten whenever an update occurred, past allocations would no longer reflect the reality of previous academic periods. The chosen versioned model guarantees that all allocations remain permanently linked to the data that was valid when they were created.

Overall, storing all domain rules in the database and maintaining multiple layout versions increases reliability, correctness, and traceability—despite the extra schema complexity it introduces.

7. Comments About the Course

The conceptual data modeling assignment, including lectures, installation and learning of the required software, practical modeling, and preparation for the workshop, has taken up all the time since the beginning of the course. But since conceptual data modeling is the first step in the data modeling process, I believe that solving the practical problem will greatly help me in my future practice.