Project Report

Data Storage Paradigms, IV1351

2025/11/18

Project members:61
[Tianle Niu, tianlen@kth.se & 2023110605@stu.hit.edu.cn]

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

This report presents the work for Seminar 1, Task 1: *Logical and Physical Model (Only Mandatory Part)*. The target of this task is to translate the given conceptual description into a relational model (the same as logical model in the video) with sufficient physical detail to be implemented as a functioning database (using SQL language to implement it). The model is documented as an ER diagram using crow's foot notation and is implemented in an SQL database that matches the model exactly.

The solution must cover the full requirement in the business requirement: courses and their static properties, course offerings per study year and study period, activity types with planning factors, planned teaching activities (planned hours per activity type for each course instance), organizational structure (departments and employees including job titles, supervisors, and department managers), and the allocation of employees to course instances and activity types with allocated hours. The database schema must enforce core integrity requirements through primary keys, foreign keys, uniqueness constraints,

and basic domain constraints. In addition, enough representative data must be inserted to enable meaningful queries and to support the following tasks.

All tasks were completed independently; no other students contributed to the design, implementation, or report writing.

2. Literature Study

Before starting the implementation, I reviewed the lecture slides on the Relational Model and SQL. My main focus was on understanding how to translate the conceptual many-to-many relationships (like employees working on multiple course instances) into physical tables (using another table as a "bridge"). The lecture on normalization was particularly helpful; initially, I considered storing the "teaching hours" directly on the employee table. However, after reviewing the Third Normal Form (3NF) rules regarding transitive dependencies, I realized this would create redundancy. I learned that I needed an associative entity (allocation) to properly link employees to activities without violating 3NF. (Actually, I found that 3NF is more convenient as a detector to find out redundancy, not the formula to design the whole schema.)

I also watched the supplementary videos on logical and physical models. This video gives a clear path to translate the conceptual model into logical one.

In addition, I learned DBeaver, because its seems is widely used in the industry and can generated ER diagram automatically, which is convenient for me in this assignment.

3. Method

This task was solved by following a requirements-driven modeling procedure like what I have learnt from videos that started from the domain description and ended in a validated logical model and a matching physical database schema.

Tools. I used *DBeaver* as the main database client throughout the work: to mainly observe the schema, to insert and inspect data, and to automatically generate the ER diagram directly from the implemented tables. Besides, I will write SQL directly because it will be more efficient when debugging. The database was implemented in *PostgreSQL*, and all SQL scripts were version-controlled and delivered via *GitHub*.

Procedure. I started by breaking down the assignment text into a requirement checklist. I used pen and paper to sketch the initial entities (Course, Employee, Department) before writing any code. Appendix A contains these original sketches.

Once the concept was clear, I moved to DBeaver to implement the schema in PostgreSQL. I built the tables incrementally, starting with independent entities like department and activity_type to avoid foreign key errors. A challenge I encountered was modeling the organizational hierarchy; I initially thought about making a separate "Supervisor" table, but after testing the schema, I realized a self-referencing foreign key on the employee table was a cleaner solution. Besides, I also hesitate about whether to use course_layout as index to bind couse_instance or not, and I realised if I do so, it will obey the 1NF. Lastly, I check the database to make sure it satisfy 3NF.

Finally, I validated the model by inserting test data that specifically targeted edge cases, such as an employee teaching multiple courses, to ensure the primary keys did not conflict.

4. Result

4.1. Repository and scripts

All SQL scripts are available in the Git repository:

`https://github.com/turturturturtur/KTHIV1351.git`

The repository contains two scripts, as required:

- `001_CreateDataBase.sql`: creates the database schema (including some debug code, you can ignore it).
- `002_InsertData.sql`: inserts the data used for validation.

4.2. Model Overview and Academic Structure

The core of the solution is the separation of static course data from specific offerings. As shown in the ER diagram (Figure 1), the `course_layout` table holds constant attributes (credits, course name), while `course_instance` represents a specific offering in a given `period` and `study_year`. This design allows the same course to be offered multiple times without duplicating data.

The ER diagram was generated directly from the physical PostgreSQL schema using DBeaver, ensuring that the documentation matches the actual code exactly.

4.3. Resource Planning and Allocation

To differentiate between "budgeted" hours and "actual" work, I implemented two distinct associative tables:

- **Planning:** The `planned_activity` table links a `course_instance` to an `activity_type` (e.g., Lecture, Lab). This stores the total hours required for that activity type.
- **Allocation:** The `allocation` table assigns specific employees to these activities.

This structure enables the system to track if the sum of allocated hours matches the planned hours. I used a specific `activity_type` table with a `factor` column to act as a controlled vocabulary, avoiding free-text errors (e.g., spelling "Lecture" differently).

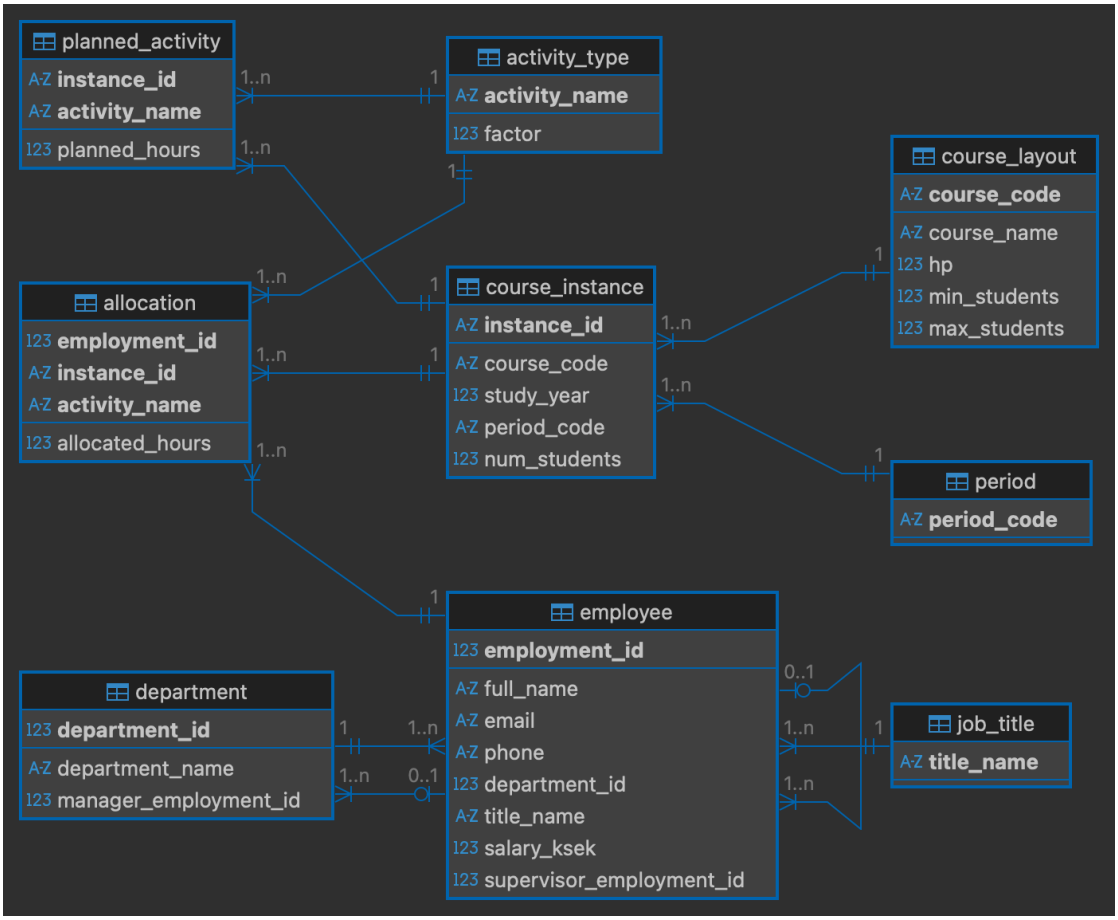

Figure 1: ER diagram of the implemented database schema.

4.4. Organizational Structure

The organizational hierarchy is modeled within the `employee` and `department` tables. Instead of creating a separate table for supervisors, I used a self-referencing foreign key (`supervisor_employment_id`) on the `employee` table. This allows for a flexible hierarchy where any employee can theoretically supervise others. Department managers are linked via the `manager_employment_id` in the `department` table.

4.5. Data Integrity and Test Data

Database integrity is enforced through a combination of constraints:

- **Keys:** Primary keys identify all entities (using surrogate keys like `allocation_id` or composite keys where appropriate). Foreign keys enforce referential integrity between all related tables.

- **Domain Constraints:** I added specific `CHECK` constraints to prevent invalid logical states, such as ensuring `allocated_hours` and `num_students` are non-negative, and that activity factors are greater than zero.

To verify the model, I populated the database with a manually constructed dataset. This dataset was designed to test specific edge cases, such as an employee (e.g., 'Paris Carbone') who acts as both a department manager and a teacher, and cases where one teacher is allocated to multiple different course instances in the same period.

5. Discussion

In this chapter, I analyze the database design and evaluate how well it meets the logical and physical modeling requirements.

5.1. Naming Conventions

I used `snake_case` for all identifiers to maintain consistency with standard PostgreSQL practices. I tried to make the names self-explanatory; for example, distinguishing between `course_layout` (the static definition) and `course_instance` (the specific offering) helps clarify the difference between a course and a specific time it runs.

5.2. Diagram Notation

The ER diagram (Figure 1) was generated directly from the database schema using DBeaver. This ensures that the diagram is 100% consistent with the actual SQL code. While DBeaver's default rendering is slightly different from hand-drawn Crow's Foot notation, the cardinalities are correct:

- One department has many employees.
- One course layout acts as a template for many course instances.
- Employees and Courses have a Many-to-Many relationship, which I resolved using the `allocation` table.

5.3. Normalization (3NF)

I designed the schema to adhere to Third Normal Form (3NF). For example, I placed the `factor` attribute in the `activity_type` table rather than in `planned_activity`. If I had stored it in the planned activity table, I would have had to update every single row if the university decided to change the factor for "Lectures," which would be an update anomaly. The only "redundancy" is strictly for performance (foreign keys), so the model satisfies 3NF.

5.4. Keys and Constraints

For the `course_instance` table, I decided to use a surrogate primary key (`instance_id`) rather than a composite key of `(course_code, year, period)`. While the composite key is the "natural" key, using it would have made the foreign keys in the `allocation` table very complex and hard to read. However, I added a `UNIQUE` constraint on those three columns to ensure we don't accidentally create duplicate course instances.

I used `CHECK` constraints for domain integrity, such as ensuring `salary_ksek` and `hours` cannot be negative.

5.5. Business Rules Not Enforced in SQL

There are some complex business rules that simple Primary/Foreign keys cannot enforce. I identified three main rules that would require Triggers or application logic to solve:

- **Student Limits:** The database allows `num_students` in an instance to be outside the `min/max` range defined in the layout. This requires a cross-table check.
- **Over-allocation:** Currently, nothing stops me from allocating 100 hours to a teacher even if the planned activity is only 20 hours. Checking `sum(allocated) <= planned` requires aggregation, which cannot be done in a standard `CHECK` constraint.
- **Manager Integrity:** A department manager should ideally be an employee of that same department. Currently, the foreign key only requires them to be *an* employee.

5.6. Derived Attributes

I avoided storing derived attributes. For example, I did not create a column for "Total Workload" in the employee table. Storing this would be risky because every time an allocation is updated, the total would need to be recalculated. Instead, this should be calculated via a `SELECT SUM()` query when needed.

6. Comments About the Course

I spent approximately 15 hours on this assignment in total. About 10 hours were used to study the lecture material (including recorded content), and about 5 hours were spent learning basic SQL usage in practice, getting familiar with DBeaver, and debugging/testing the database implementation.

Overall, the step-by-step modeling material was helpful for turning requirements into a concrete schema. One suggestion for future offerings would be to provide a short checklist or a minimal example report structure for Task 1, so that students can better align their writing with the expected level of detail and evidence.

A. Appendix: Original Handwritten Design Notes

This appendix contains scans of my original handwritten design notes created during the modeling process. The notes document my reasoning when translating requirements into entities, relationships, keys, and constraints, and are included as supporting evidence of independent work and understanding.

For convenience, I wrote these notes in Chinese to support my thinking and inference. If needed, I can translate them into English.

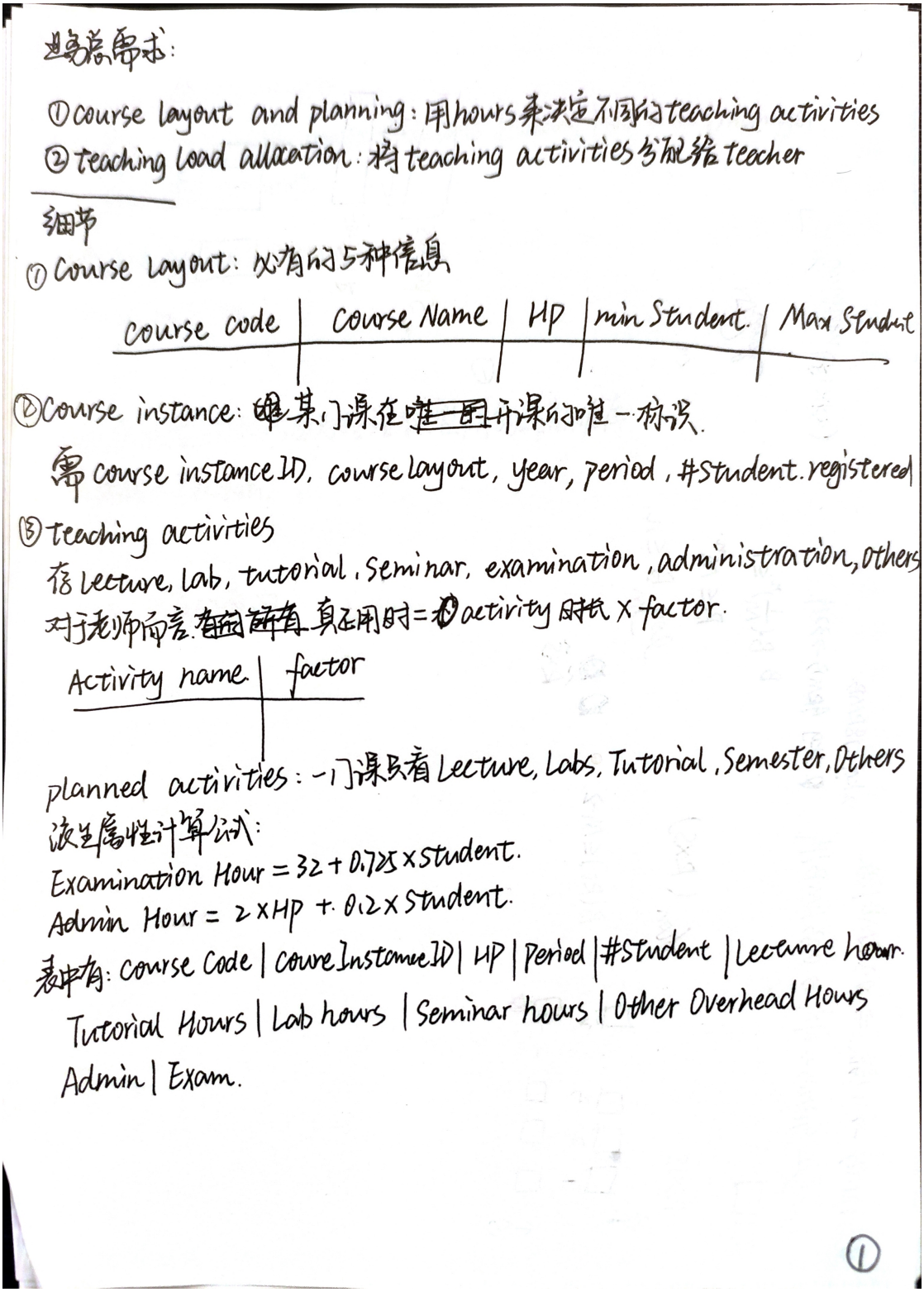

Figure 2: Handwritten design notes, page 1.

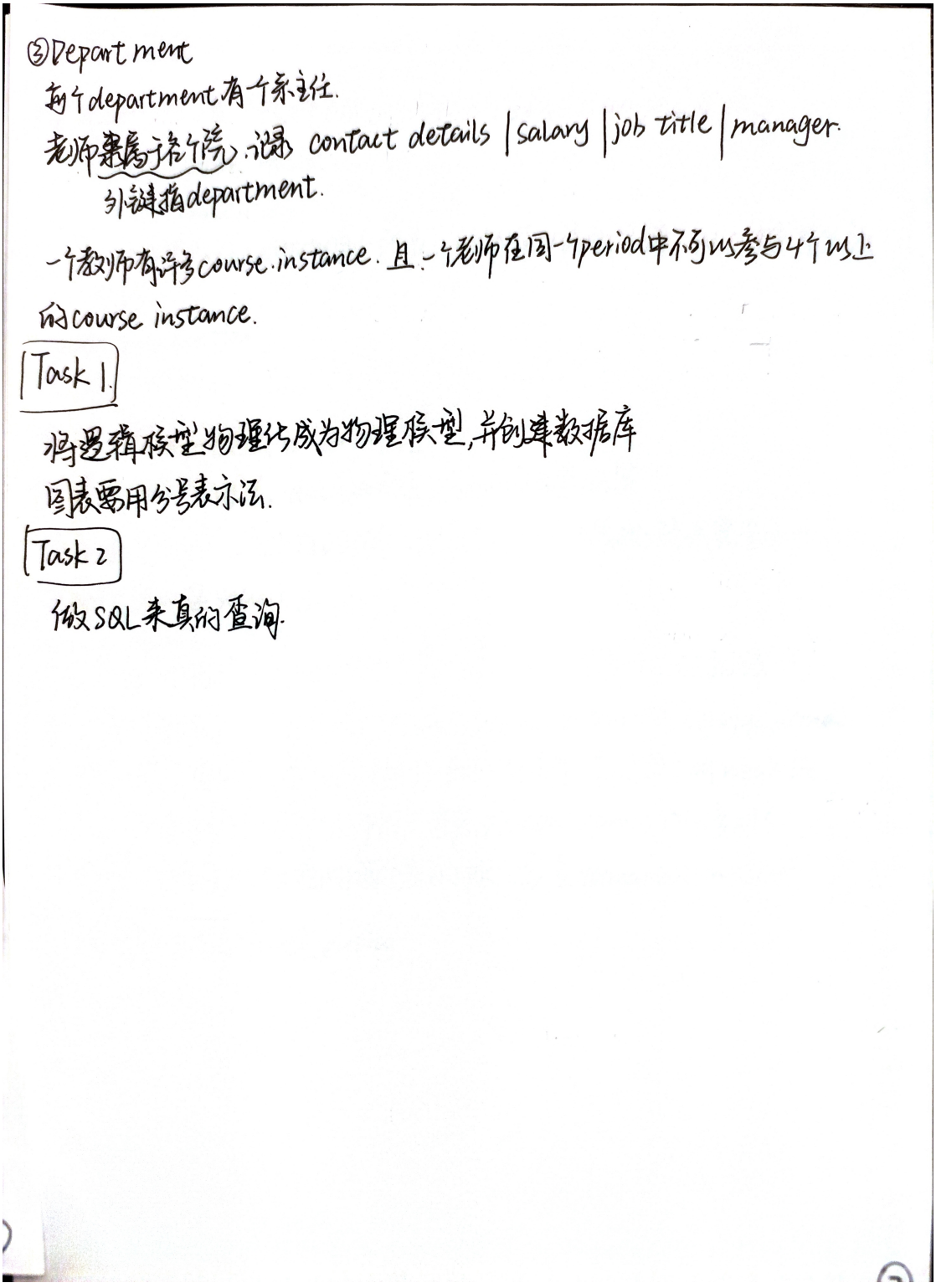

Figure 3: Handwritten design notes, page 2.

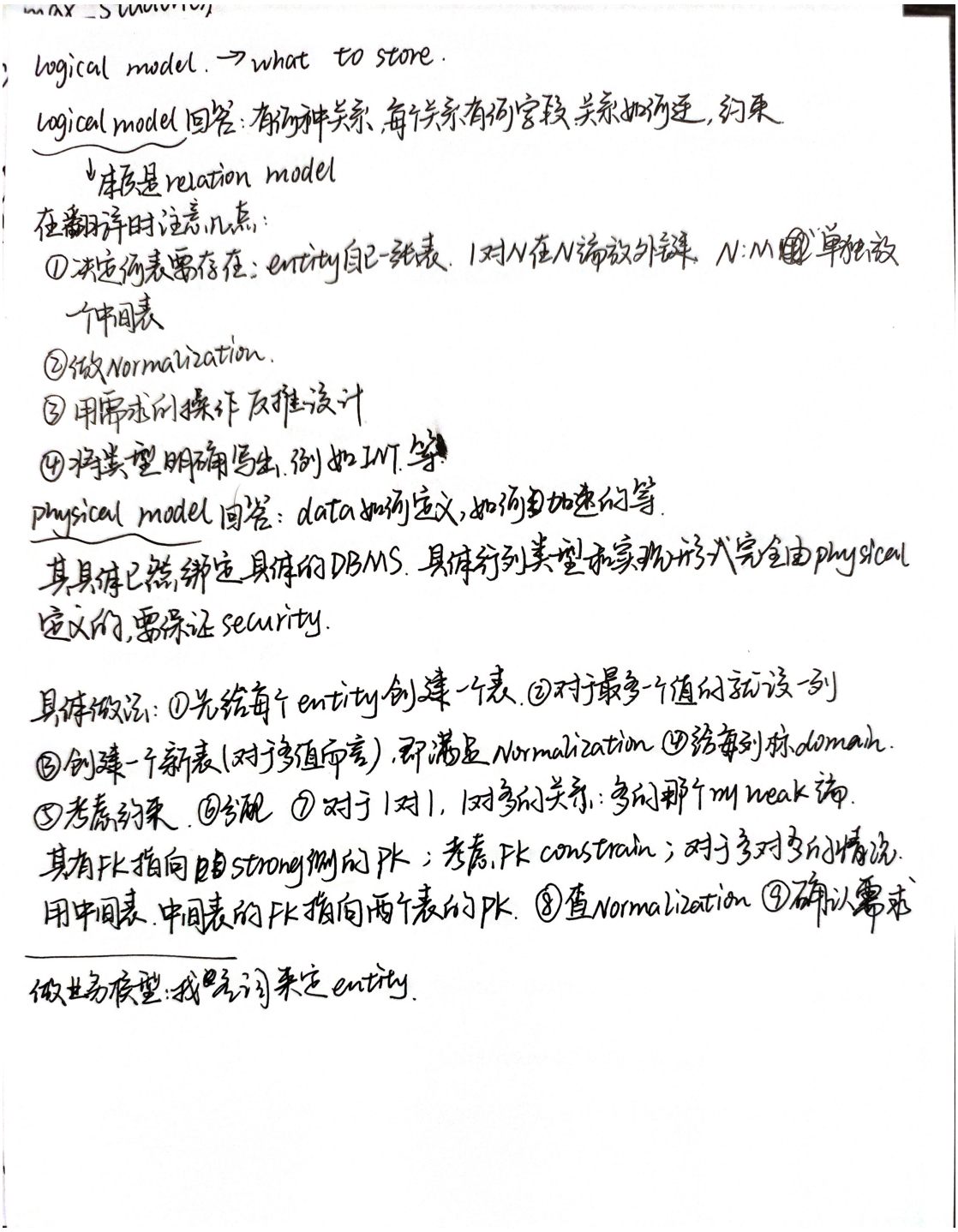

Figure 4: Handwritten design notes, about logical models.

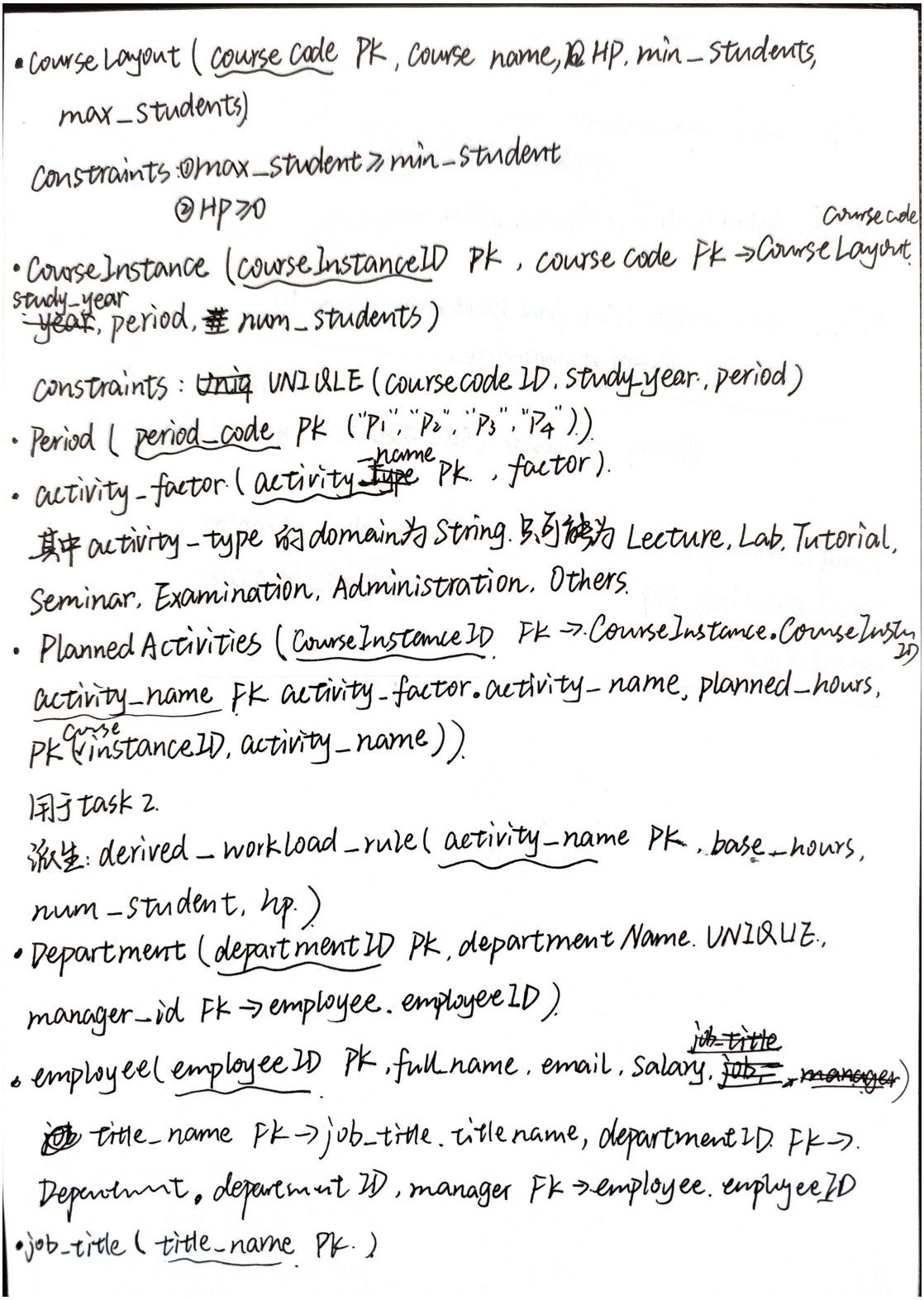

Figure 5: Handwritten design notes, about logical models.

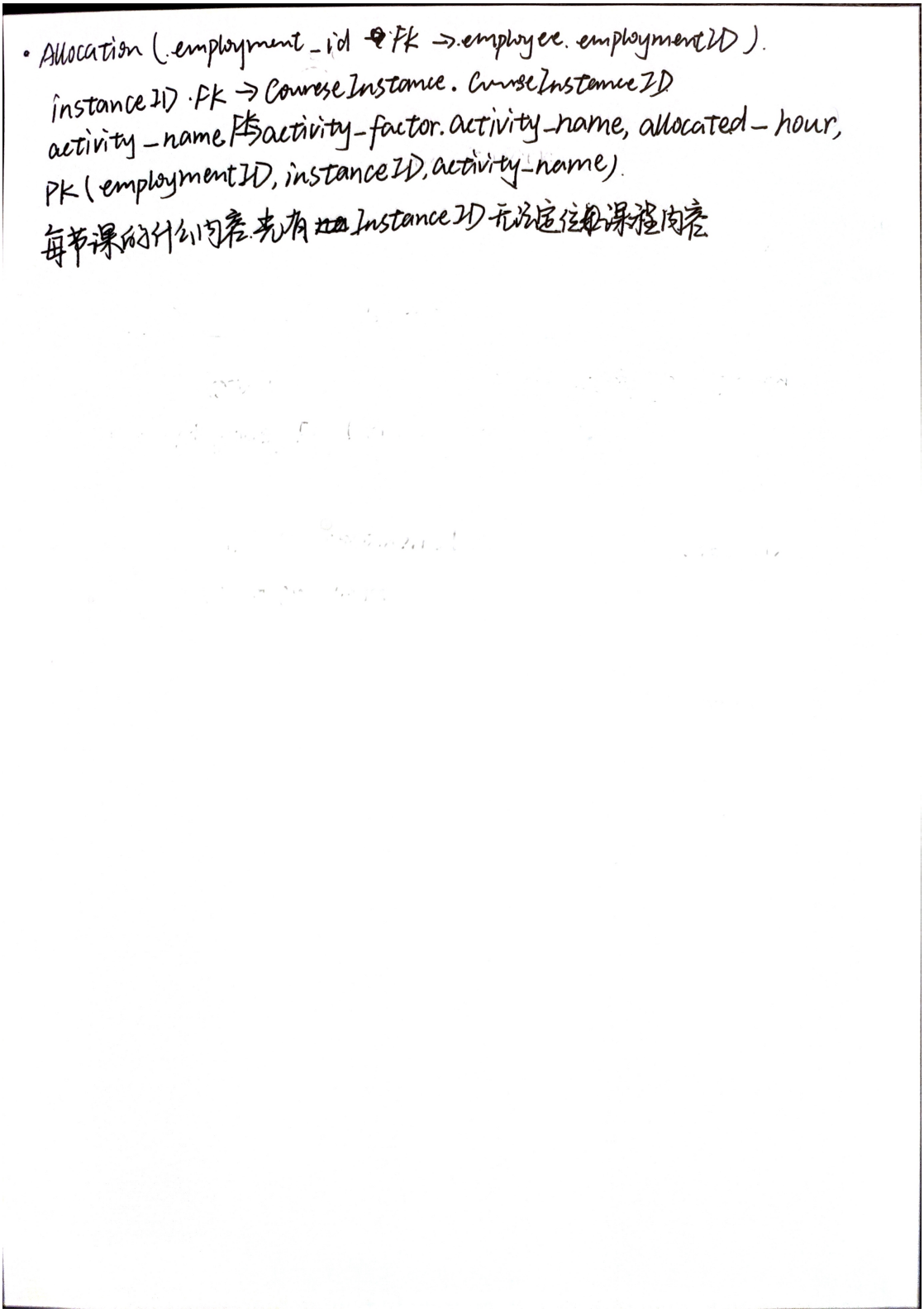

Figure 6: Handwritten design notes, about logical models.