Project Report

Data Storage Paradigms, IV1351

Lana Ryzhova

lana_lana67@ukr.net

Ångermannagatan 1B lgh 1003

162 64 Vällingby

2024-11-20

Data Storage Paradigms, IV1351 Project Report

Task 2, Logical and Physical Model

1 Introduction

1. **The purpose of this report is:**

1.1 Transforming a conceptual model into a logical model with enough physical aspects to create a database.

1.2 Creation of a logical and physical model of the Soundgood music school database.

1.3 Transforming a model into a functioning database using a relational database and query languages.

2. **The task was solved independently.**

2 Literature Study

1. Lectures on normalization and logical and physical models from the book "Fundamentals of Database Systems (7th Edition) by Ramez Elmasri and Shamkant B. Navathe" were read and understood.

2. Videos on logical and physical models were viewed.

3. Using the PostgreSQL program, a database was created and filled with test data.

4. Using Data Modeling using the Entity–Relationship (ER) Model, with using crow's Foot Notation, the task of creating logical and physical models of the Soundgood music school was solved.

5. The report and solution to this problem have been posted in my Git repository.
   https://github.com/Lana-1167/KTX-1167.git

   The data for the second seminar is in the folder
   https://github.com/Lana-1167/KTX-1167.git /seminar2/

6. The document tips-and-tricks-task2.pdf was read and understood.

3 Method

Basic Components of Logical Data Modeling:

1. **Entities, attributes, and relationships.** These components retain their original meaning and purpose from the conceptual data model.

   1.1 Entities represent key objects or concepts in a domain, such as Student, Instructor, Musical Instrument, etc.

   1.2 Attributes define properties of objects, such as Student Name, Person number, Data of rent instruments, etc.

   1.3 Relationships represent connections between objects, such as a Student placing multiple orders for different lessons on different instruments,

Musical Instruments belonging to different brands and having different types, etc.

2. **Data types:** Assign specific data types to each attribute, defining the type of information it can store, such as integers, strings, or dates.

3. **Constraints:** Define rules or constraints that the data stored in attributes must adhere to, such as uniqueness, referential integrity, or domain constraints.

Procedure for creating a logical data model.

Steps to create a logical data model include several stages:

1. **Refine entities, attributes, and relationships:** Review and update components migrated from the conceptual data model, ensuring that they accurately reflect the intended business requirements. Make the model more efficient where possible, for example by identifying reusable entities or attributes.

2. **Defining data types and constraints:** You must assign appropriate data types to each attribute and specify any constraints that need to be applied to ensure data consistency and integrity.

3. **Normalization of the logical data model.** Normalization techniques should be applied to eliminate redundancy and improve the efficiency of the data model. It should be verified that each entity and its attributes comply with the requirements of the various normal forms (1NF, 2NF, 3NF, etc.).

Once the logical data modeling process is complete, the resulting model is ready for the final stage of physical data modeling.

The main components of physical data modeling are:

1. **Tables:** These represent the actual storage structures for entities in the data model, with each row in a table corresponding to an instance of an entity.

2. **Columns:** Correspond to attributes in the logical data model, defining the data type, constraints, and other database-specific properties for each attribute.

3. **Indexes:** Define additional structures that improve the speed and efficiency of data retrieval operations in tables.

4. **Foreign keys and constraints:** Represent relationships between tables, ensuring that referential integrity is maintained at the database level.

The steps for creating a physical data model include several stages:

1. **Select a DBMS:** Select a specific database management system (e.g. PostgreSQL, MySQL, or SQL Server) on which the physical data model will be implemented. This choice will determine the available model functions, data types, and constraints.

2. **Map logical objects to tables.** Create tables in the selected DBMS to represent each object in the logical data model and their attributes as columns in the table.

3. **Define indexes and constraints.** Create all necessary indexes to optimize query performance and define foreign key constraints to ensure referential integrity between related tables.

4. **Create database objects.** Using a data modeling tool or manually writing SQL scripts, create actual database objects such as tables, indexes, and constraints based on the physical data model.

Physical data modeling is the final stage of the data modeling process, where the logical data model is transformed into a real implementation using a specific database management system (DBMS) and technology.

The logical and physical model was created using Data Modeling using the Entity – Relationship (ER) Model, using crow's Foot Notation.

4 Result

The database was created using the open source database management system **PostgreSQL**. The database management tool used in the report to create and modify databases, insert data, and query data is the GUI tool **pgAdmin**.

The logical and physical model was created using Data Modeling using the **Entity–Relationship (ER) Model**, using **crow's Foot Notation**.

All links in this notation are binary and represent lines connecting entities. Each end of the link must be defined by a name and a degree of multiplicity, i.e. one or more objects participate in the link. The degree of multiplicity is also called the cardinal number.

When there are multiple possible keys, one of them is designated as the primary key, and the others are called alternate keys. In this notation, a data model can contain entities, supertypes and subtypes, as well as recursive relationships that link an entity to itself.

Remark from Leif Lindbäck from 27 Nov at 11:06

It's very hard to read the model. It gets a bit blurred when I zoom in, and the lines are crossing and overlapping a lot.

You can always open the diagram source. This is the file log_fis_shema.pgerd on github. Unfortunately, 30 tables on A4 report format do not fit in a readable form.

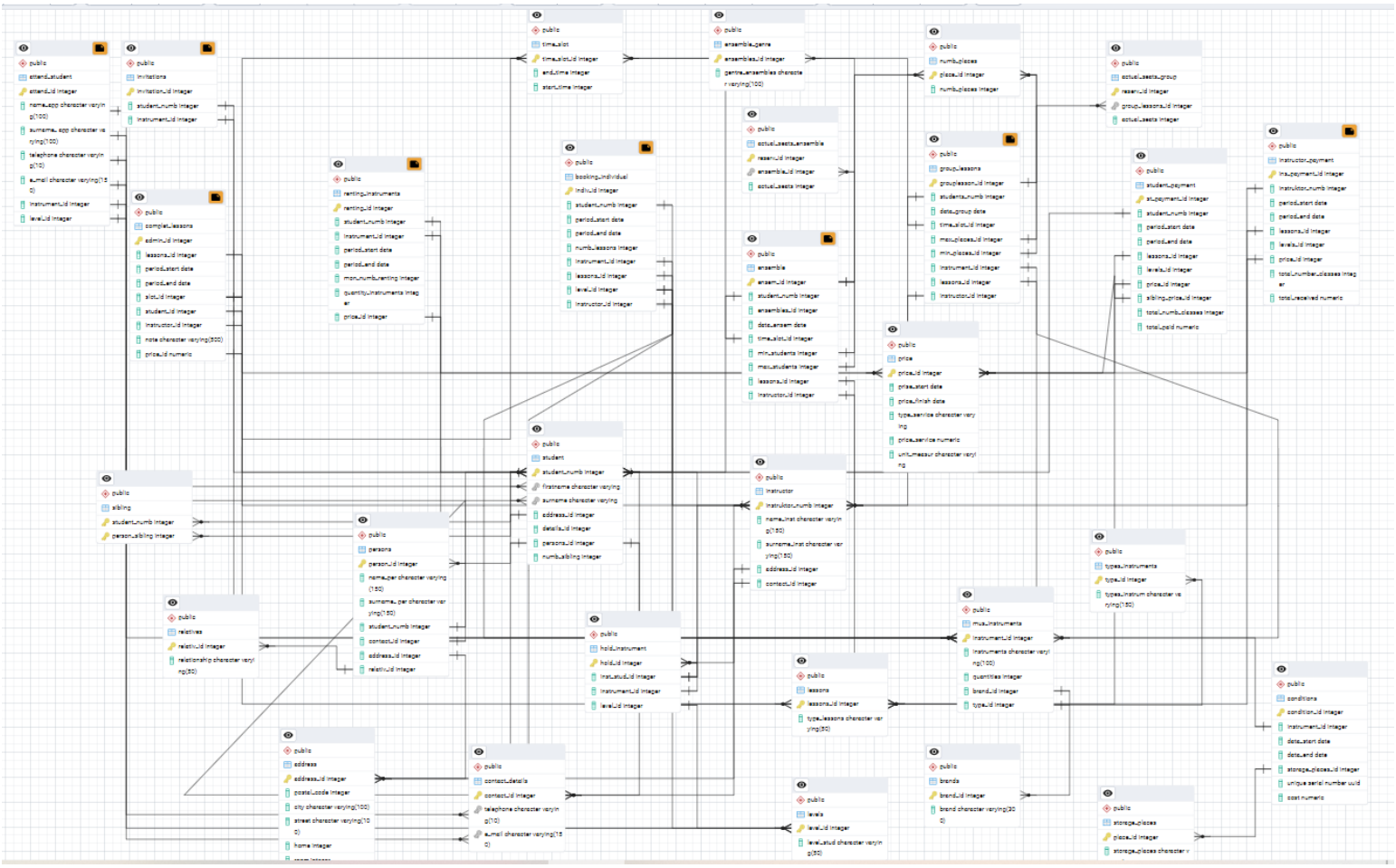
The logical and physical model is posted in the Git repository.

https://github.com/Lana-1167/KTX-1167.git/

All necessary data is in the folder

https://github.com/Lana-1167/KTX-1167.git /seminar2/

Data Modeling using the Entity–Relationship (ER) Model, with using crow's Foot Notation



To generate the data, an online generator was used https://generatedata.com/. Since the data must be consistent, some of the data was created manually.

The script that creates the database and the script that inserts the data are posted in the Git repository.

https://github.com/Lana-1167/KTX-1167.git/

5 Discussion

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

1.1 The PostgreSQL documentation describes the recommended way to name tables in PostgreSQL:

Use clear plural nouns, e.g. "levels", "students".

Views use "v_" as a naming prefix, materialized views use "mv_" as a naming prefix, and temporary tables use "tmp_" as a naming prefix.

1.2 How Not to Name Tables in PostgreSQL:

Abbreviate or use singular, such as "lev" or "stud".

Thus, the naming conventions are respected. All names are self-explanatory.

2. Is the crow foot notation correctly followed?

Yes, the notation is correct.

The logical and physical model was created using Data Modeling using the Entity–Relationship (ER) Model, using crow's Foot Notation.

All relationships in this notation are binary and represent lines connecting entities. Each end of the relationship must be given a name and a degree of multiplicity, i.e. one or more entities participate in the relationship. When there are several possible keys, one of them is designated as the primary key, and the others are called alternate keys.

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

The model is in 3NF.

The process of sequential transition to complete decompositions of database files is called normalization of database files, the main goal of which is to eliminate duplication of information and loss of attached records.

A relation is called normalized or reduced to 3NF if the relation is in 2NF and none of its non-key fields is functionally dependent on any other non-key field.

For example, 3NF Model after applying normalization, may look like this:

1. **Student**

   Student ID (PK), Other Student Attributes...

2. **Relative**

   Relative ID (PK), First Name, Last Name, Contact, Address

3. **Student Relative**

   Student ID (FK), Relative ID (FK), Relationship

Conclusion

The suggested normalization reduces redundancy and ensures adherence to 3NF principles.

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

Table relevance is the degree of compliance, relevance and reliability with the requirements for creating a database, the purpose of which is to simplify the processing of information and business operations for the Soundgood music school. **Yes, all tables are relevant. All necessary tables are present.**

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, there are columns for all the data that needs to be stored. All relevant column constraints and foreign key constraints are specified. All column types can be motivated.

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

Yes, can the choice of primary keys be motivated. Primary keys are unique.

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

Yes, all relationships are relevant. All necessary relationships are present. Cardinality is the uniqueness of the data values contained in an attribute. Yes, cardinality is correct.

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

Yes, all tasks listed in the project can be completed.

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

Yes, all business rules and constraints that are not visible on the diagram are explained in plain text. The diagram itself is posted in the Git repository. https://github.com/Lana-1167/KTX-1167.git

10. Are there attributes which are calculated from other attributes and then written back to the database (derived attributes)? If so, why? Records of student fees and instructor payments might be examples of such attributes.

Yes, there are attributes that are calculated from other attributes and then written back to the database. These are called derived attributes. Student dues and teacher payments records are examples of such attributes.

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

Yes, tables (or ENUMs) are always used for constants such as skill levels (beginner, intermediate and advanced) instead of free text.

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

Yes, the report explains the method and the result. Is there a discussion that is relevant?

Part of the highest rating

1. **Ensure that the models created in this task contain all the data required by Soundgood and do not require a database-based application to manage any data at all.**
   For example, it might be tempting to create a database that does not store the number 2 that is mentioned in the text, each student can rent up to two specific instruments, and instead hard-code the number 2 into the application that calls the database.

This task can be solved by creating a **renting_check()** trigger function for the **renting_instruments** table. To check the correctness of the data in the **renting_instruments** table, you need to use the **BEFORE** trigger.

Thus, we create a trigger that calls the **renting_check()** trigger function for each modified row of the **renting_instruments** table:

```
CREATE FUNCTION renting_check() RETURNS trigger AS $renting_check$
  BEGIN

IF NEW.quantity_instruments IS NULL THEN
      RAISE EXCEPTION '% You need to specify the number of musical instruments from 0 to
2', NEW.quantity_instruments;
    END IF;

    IF NEW.quantity_instruments > 2 THEN
      RAISE EXCEPTION '% The number of musical instruments should not be more than 2',
NEW.quantity_instruments;
    END IF;

  RETURN NEW;
  END;
$renting_check$ LANGUAGE plpgsql;


CREATE OR REPLACE TRIGGER renting_check BEFORE UPDATE OF quantity_instruments OR
INSERT ON renting_instruments
  FOR EACH ROW EXECUTE FUNCTION renting_check();
```

2. **This problem involves accounting for the fact that prices can change. For example, say that a student takes a certain type of lesson, and that the price of that lesson changes before the student has paid. In this situation, it should be possible to find the cost of the lesson when the student took it, not just the current cost, since the student must pay the cost that was in effect when the lesson was taken. You need to solve this problem by creating a model that allows you to add a new price when the price changes, rather than updating the existing price. This also means that the model should be able to determine which of all the stored prices for a particular lesson is currently valid.**
   TIP: The solution to this problem often involves storing the time when a certain value was inserted into the database. However, you don't have to do this, there are other possible solutions.

To solve this problem, we save the time of inserting the change in the lesson cost in the **price** table in the database. To do this, we add the **price_start, price_finish** attributes to the table.

5.17 Price (tab. price)

price_id	price_start	price_finish	lessons_id	levels_id	type_service	price_service	unit_measur
3	2000-01-01	2024-09-01	1	1	Individual beginner	100	SWK
9	2024-09-01	2050-01-01	1	1	Individual beginner	500	SWK

This problem can be solved by creating a trigger function **update_price_id()** for the **complet_lessons`** table. The trigger will intercept the moment the cost of a lesson changes in the **price** table and update the **price_id** field in the **complet_lessons`** table.

Thus, we create a trigger, **AFTER INSERT OR UPDATE** for the trigger function **update_price_id()**:

```
CREATE OR REPLACE FUNCTION update_price_id()
RETURNS TRIGGER AS $$

BEGIN
  -- Checking if the value needs to be updated

 IF NEW.price_id <> OLD.price_id THEN

UPDATE complet_lessons`
     SET price_id = (
       SELECT b.price_id
       FROM price a
       JOIN complet_lessons` b ON a.lessons_id = b.lessons_id AND a.levels_id = b.levels_id
       WHERE a.price_finish = b.period_start
       LIMIT 1  -- Limit the result to just one line
     )
  WHERE price_id <> NEW.price_id;

RAISE EXCEPTION '% The student must pay the fee on the date the lesson was taken!',
OLD.price_id;
    END IF;

  RETURN OLD;
END;
$$ LANGUAGE plpgsql;



CREATE TRIGGER update_price_id_trigger
AFTER INSERT OR UPDATE ON complet_lessons`
FOR EACH ROW
EXECUTE FUNCTION update_price_id();
```

Remark from Leif Lindbäck from 27 Nov at 11:06

You haven't stored the max time an instrument can be rented (12 months) in the database. That must also be done to get the higher grade points.

The database contains a table called renting_instruments. The description can be found on page 22 of "Project Report Task 1, Conceptual Model"

4.20 Renting Insrtruments (tab. renting_instruments)

renting_id	student_numb	instrum_id	period_start	period_end	quantity_instruments	price_id
PRIMARY KEY, UNIQUE, NOT NULL	NOT NULL, FOREIGN KEY	NOT NULL, FOREIGN KEY	NOT NULL	NOT NULL	NOT NULL	NOT NULL, FOREIGN KEY
1	117	1	2024-01-09	2025-01-09	2	1
2	117	2	2024-10-09	2024-05-09	1	2

To solve the problem, it would be easier to update the value if it was stored in the table. But, since the problem statement says that "Soundgood wants to have a high level of flexibility", and it is possible that over time, the maximum rental period for musical instruments will change. Therefore, we will implement a check that the rental of an instrument can last no more than 12 months. To do this, we will add a trigger to the database. The trigger will check whether the rental period of a given instrument exceeds 12 months. Such a check corresponds to the condition and will provide flexibility if the rental period changes over time.

Solution:

Checking the lease duration: When trying to update or insert a record into the renting_instruments table, the trigger should check the difference between the lease start and end dates. If the difference is more than 12 months, the operation should be rejected.

The trigger will fire on INSERT and UPDATE operations on the renting_instruments table.

```
-- Create a trigger function to check the lease duration
CREATE OR REPLACE FUNCTION check_rental_duration()
RETURNS TRIGGER AS $$
BEGIN
  -- We check that the difference between the start and end dates of the lease does not exceed 12 months
  IF EXTRACT(YEAR FROM AGE(NEW.period_end, NEW.period_start)) * 12 + EXTRACT(MONTH FROM AGE(NEW.period_end, NEW.period_start)) > 12 THEN
    RAISE EXCEPTION 'Rental period cannot exceed 12 months.';
  END IF;

  -- If the check is passed, return NEW (data to insert or update)
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create a trigger that calls a function when a record is inserted or updated in the renting_instruments table
CREATE TRIGGER rental_duration_check
BEFORE INSERT OR UPDATE ON renting_instruments
FOR EACH ROW
EXECUTE FUNCTION check_rental_duration();
```

Explanation:

AGE(NEW.period_end, NEW.period_start) calculates the interval between two dates.

EXTRACT(YEAR FROM AGE(...)) * 12 extracts the number of years from an interval and multiplies it by 12 to convert it to months.

EXTRACT(MONTH FROM AGE(...))extracts the number of months remaining after accounting for full years.

The result is added up and if the number of months exceeds 12, an exception is triggered.

Example of work:

Attempt to insert a lease exceeding 12 months:

```
INSERT INTO renting_instruments (renting_id, student_numb, instrum_id, period_start, period_end, quantity_instruments, price_id)
VALUES (4, 107, 1, '2024-01-09', '2025-04-09', 2, 1);
```

In this case, the trigger will throw an exception:

```
ERROR: Rental period cannot exceed 12 months.
```

Attempt to insert a lease within 12 months:

```
INSERT INTO renting_instruments (renting_id, student_numb, instrum_id, period_start, period_end, quantity_instruments, price_id)
VALUES (4, 107, 1, '2024-01-09', '2024-12-09', 2, 1);
```

This request will be successful because the lease lasts for 12 months.

Conclusion

Error protection: This constraint will prevent you from mistakenly entering a lease that lasts longer than the allowed period (e.g. 12 months) without having to manually track this data.

Business rules automation: This trigger ensures that the data in the table always complies with the business rules.

3. Discuss the advantages and disadvantages of storing all data in a database, as done here, rather than having some data in the application.

Storing data in a database versus hard-coding values directly in the application is an important architectural decision. Both approaches have their advantages and disadvantages. Let's look at them in more detail using the example of the number 2, which limits the number of instruments a student can rent.

1. Benefits of Storing Data in a Database:

1.1 Flexibility and scalability:

If the limitation changes in the future (for example, students can rent 3 instruments instead of 2), it will be possible to update the data in the database without changing the application code. This simplifies the support and adaptation of the system.

1.2 Data uniformity:

Storing such constraints in the database allows you to ensure data consistency between different applications or system components. For example, if you have multiple microservices or applications that use the same database, changing the logic in one place (the database) will immediately affect all applications.

1.3 Support for changes and configurations:

In case of business logic changes (for example, if rental policies change at the level of different student programs or categories of students), only the records in the database can be changed, without the need to change the source code of the application.

1.4 Security and access control:

Database security mechanisms (such as roles and permissions) allow you to control who can change sensitive data (such as rental restrictions), which improves security compared to hard-coding that data in the application.

1.5 Reporting and analytics support:

Storing all data in a database makes it easy to gather statistics, create reports, and analyze data, which can be difficult to do with hard-coded values in an application.

2. Disadvantages of storing data in a database:

2.1 Difficulty in management:

Storing flexible values in a database may require additional complexity in the design of the data structure, as it will require handling more complex queries and additional tables to store this data. For example, a separate table may be required for configurations or constraints, which requires additional database operations.

2.2 Performance:

If your system frequently accesses the database for settings or small values (such as limits), this can impact performance because each database access is a network request, which can be slow compared to reading data from the application's local memory.

2.3 Database dependency:

If you have a hard dependency on data stored in a database, this can lead to problems if the database becomes unavailable or if connection errors occur, which will affect the functionality of the application.

2.4 Less predictability in behavior:

If the values are stored in a database, data consistency issues may arise, especially if the database has multiple update sources or incorrect synchronization.

3. Benefits of Hard Coding in an Application:

3.1 Simplicity and productivity:

Hard-coding values in an application can be easier to implement in small systems or if the values do not change frequently. The application can work quickly and efficiently with locally available values, without having to query the database. The application can quickly and without latency access this data from memory, which can be useful for small and high-performance systems.

3.2 Less dependencies:

The application does not depend on external data sources, such as a database. In case of a database crash or connection errors, the application will still function (if it does not depend on other data).

3.3 Simplifying development:

If the values are known in advance and are unlikely to change, then hard coding can simplify the design of the system, since you do not need to design a database structure to store these values.

4. Disadvantages of Hard Coding in an Application:

4.1 Lack of flexibility:

If a limitation (such as the number of instruments a student can rent) changes, you'll need to change your application code and possibly rebuild and deploy it, which slows down the process of making changes and scaling.

4.2 Difficulties with updates:

If the values are hard-coded, you will need to update the app on all servers or devices where it is installed. This can be a labor-intensive process, especially if the app is used by many users or in different locations.

4.3 Problems with support:

Maintaining and extending the system can become difficult because business logic (such as rent restrictions) will be scattered throughout the code. This increases the likelihood of errors and makes the system less convenient for scaling.

4.4 Lack of centralized control:

If you need to change the logic or rental constraint, you will need to change the code at all levels of the system, rather than just updating records in one database. This makes the system less flexible and more labor-intensive to maintain.

5. When to Use Each Approach:

Storing in a database is appropriate when values may change over time, when they need to be accessible to different parts of the system, or when the system requires flexibility in settings and configurations.

Hard coding is justified if the values are fixed, will not change, and if you have the resources and desire to manage the system through code updates rather than through the database.

6. Summary:

In general, storing values in the database is preferable, especially for business logic that may change or is important for centralized management. Hard coding is limited to situations where such values are fixed and rarely change, or in small applications where simplicity and performance are important.

6 Comments About the Course

All the time after the first seminar was spent on solving the task on logical and physical data modeling, including lectures, creating a database and filling it with the necessary data, practical modeling and preparation for the seminar. I believe that solving such a practical task will greatly help me in the future.