# Project Report

Data Storage Paradigms, IV1351

**Lana Ryzhova**

lana_lana67@ukr.net

Ångermannagatan 1B lgh 1003

162 64 Vällingby

2024-12-01

# Data Storage Paradigms, IV1351 Project Report

## Task 3, SQL

# 1 Introduction

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

1.1 Describe and explain the basic concepts, principles, and theories in the fields of data/databases/data warehousing, information administration, and database design.

1.2 To analyze business and create reports, you need to create OLAP, online analytical processing, queries and views.

1.3 Analyze the performance of one of the queries using EXPLAIN ANALYZE.

1.4 Ensure that the database contains enough data to test that all queries work as expected.

2. **The tasks was solved independently.**

# 2 Literature Study

1. The lecture on SQL from the book "Fundamentals of Database Systems (7th Edition) by Ramez Elmasri and Shamkant B. Navathe" was read and understood.

2. Relational databases and query languages in the PostgreSQL program were used.

3. The report and solution to this problem have been posted in my Git repository.

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

All necessary data is in the folder

https://github.com/Lana-1167/KTX-1167/seminar3/

4. The document was read and understood tips-and-tricks-task3.pdf.

# 3 Method

### 3.1 In the Method chapter of your report, indicate which DBMS you are using, which tool you are using to develop your SQL queries, and how you have verified that your SQL queries work as intended.

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 pgAdmin GUI.

### 3.2 Describe and explain the basic concepts, principles and theories in the field of data/databases/data warehousing, as well as in the field of information administration and database design

In the field of data, databases and information management, there are several key concepts, principles and theories that form the basis for the design, management and use of databases. These principles cover various aspects: from storing and retrieving data to ensuring the integrity, security and optimization of database work. The main ones are:

**1. Data Models**

**Data models** define how data is organized, stored, and retrieved in a database. The main data models are:

- **Relational Model:** Based on the representation of data in the form of tables (relations), each row of the table is a record, and the columns are attributes.

Relationships between tables are defined using keys (primary and foreign). Example: MySQL, PostgreSQL, Oracle.

- **Hierarchical Model:** Represents data as a tree structure, where each element (or node) can have only one parent element, but can have multiple children. Example: IBM's IMS.
- **Network Model:** More flexible than hierarchical, allows one element to have multiple parents and multiple children, which makes the structure more complex and multi-connected. Example: IDMS.
- **Object-Oriented Model:** In this model, data is represented as objects, similar to objects in object-oriented programming. Example: PostgreSQL with an extension for object data types.
- **Document-Oriented Model:** Stores data as documents, usually in JSON or BSON format. Used in NoSQL databases. Example: MongoDB.

### 2. Data normalization

**Normalization** is the process of organizing data in a database to minimize redundancy and eliminate anomalies (e.g. updates, insertions, and deletions). The main forms of normalization are:

- **First Normal Form (1NF):** All attributes in a table must be atomic (inseparable).
- **Second Normal Form (2NF):** 1NF must be satisfied, and all non-key attributes must depend on the entire composite key (if any).
- **Third Normal Form (3NF):** 2NF must be satisfied, and all non-key attributes must be independent of each other (non-transitive dependencies).
- **Boyce-Codd Normal Form (BCNF):** This is a strict version of 3NF, where for every determinant in the table there must be a key dependency.

Normalization helps improve the structure of your data and makes it more logical, but sometimes over-normalization can lead to increased query complexity and poor performance.

### 3. Transaction Theory

**Transaction theory** describes how to ensure the integrity of a database when performing operations that can be interrupted. Basic principles:

- **ACID** (Atomicity, Consistency, Isolation, Durability):
- **Atomicity**: A transaction either executes completely or does not execute at all.

- **Consistency**: A transaction moves the database from one consistent state to another.
- **Isolation**: Each transaction must be isolated from others so that they do not affect each other.
- **Durability**: Once a transaction executes, changes are saved to the database, even if it fails.

**Transactions and Locks:** To ensure isolation and integrity, locking mechanisms are often used to prevent multiple users from simultaneously modifying the same data**.**

### 4. Backup and restore data

**Data backup** is the process of creating copies of data to protect against loss. The main approaches are:

- **Full backup**: Copies the entire database.
- **Incremental backup**: Copies only the changed data since the last backup.
- **Differential backup**: Copies the changes since the last full backup.

The process of restoring data after a failure must be fast and reliable. It is important to ensure regular and automated backups.

### 5. Performance and Optimization

Database performance is important to ensure fast system response. Basic optimization techniques:

- **Indexes**: Indexes help speed up searching for data in the database. However, they can slow down insert and update operations because the indexes must be kept up to date.
- **Caching**: Keeping frequently accessed data in memory can help speed things up considerably.
- **Normalization and Denormalization**: Denormalization (deliberately adding redundant data to tables to speed up queries) is sometimes used to improve performance in situations where read speed is more important than maintaining strict normalization.
- **Sharding**: Dividing data across multiple servers or databases for scalability.

## 6. Data security

Data security is the protection of data from unauthorized access, loss, and damage. Key concepts:

- **Encryption**: Using cryptographic methods to protect data, both at rest (disk-level encryption) and in transit (encrypted connections, such as SSL/TLS).
- **Access Management**: Using a role-based model to manage access rights to data. Examples: creating roles for administrators, users, etc.
- **Auditing and Monitoring**: Regularly monitoring user activity and changes to the database helps identify potential threats and prevent attacks.

## 7. Database design

Database design is the process of creating a database schema that will support business requirements. It includes the following steps:

- **Requirements Gathering**: Analyzing user and business needs to understand what data needs to be stored.
- **Conceptual Design**: Creating a high-level data model that describes entities and their relationships without binding them to a specific technology.
- **Logical Design**: Defining the structure of tables, relationships, attributes, and keys.
- **Physical Design**: Considering the storage and access features of the data, including the choice of database technologies and the physical organization of the data.
- **Implementation and Testing**: Once designed, the database schema is implemented and tested to verify correctness and performance.

## 8. Database types

- **Relational Database Management Systems (RDBMS)**: Standard databases that use tables and SQL to work with data. Examples: MySQL, PostgreSQL, Oracle.
- **NoSQL Database Management Systems**: Databases designed to work with large volumes of unstructured data. Examples: MongoDB (document-oriented), Cassandra (column-oriented), Neo4j (graph-oriented).
- **Hybrid Database Management Systems**: Systems that support both relational and NoSQL data models. Example: PostgreSQL with the JSONB extension.

### Conclusion

Database design and administration is a multifaceted field that encompasses both theoretical concepts and practical principles. Understanding data models, normalization principles, transactions, security, and performance helps create reliable and efficient systems that can support large volumes of data with high availability and security.

# 4 Result

**Task 1:**

**In the Result chapter of your report, include a link to a git repository where you have stored a script with all queries. Also explain each query and show that all queries work as intended by including the output of each query. The git repository must also contain the scripts that create the database and insert data. It shall be possible to test your solution by executing first the script that creates the database, then the script that inserts data, and finally any of the queries created in this task.**

The script that creates the database, the script that inserts the data, and the script with all the queries are posted in the Git repository.

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

All necessary data is in the folder KTX-1167 /seminar3/

**Request №1**

**Show the number of lessons given per month during a specified year. This query is expected to be performed a few times per week. It shall be possible to retrieve the total number of lessons per month (just one number per month) and the specific number of individual lessons, group lessons and ensembles (three numbers per month). It's not required that all four numbers (total plus one per lesson type) for a particular month are on the same row; you're allowed to have one row for each number as long as it's clear to which month each number belongs. However, it's most likely easier to understand the result if you do place all numbers for a particular month**

**on the same row, and it's an interesting exercise, therefore you're encouraged to try. Table 1 below is an example result of such a query, illustrating the expected output.**

**Table 1. Expected output for query 1. This example is only meant to illustrate the expected rows and columns. It's perfectly fine to change text formatting, and also to change the values.**

| **Month** | **Total** | **Individual** | **Group** | **Ensemble** |
|---|---|---|---|---|
| **Oct** | **2** | **1** | **0** | **1** |
| **Nov** | **3** | **0** | **2** | **1** |
| **Dec** | **10** | **4** | **4** | **2** |

To execute a query that retrieves the number of lessons by type and month, you can use a SQL query in PostgreSQL with aggregation by month and lesson type. We will use the TO_CHAR function to extract the month from the date and group the data by month and lesson type.

There is a table with information about completed lessons, which is called complet_lessons, and has the following columns:

- admin_id — unique lesson identifier
- period_start period_end — lesson date
- lesson_id — link to the lessons table, which contains the lesson type (e.g., "individual" has lessons_id=1, "group" has lessons_id=2, "ensemble" has lessons_id=3)

**SQL- request:**

```
SELECT
    TO_CHAR(period_start, 'Mon') AS month,                     -- Extracting the month in abbreviation
                                                                  format
    SUM(COUNT(*) FILTER (WHERE lessons_id = 1)) OVER ()        -- Number of individual lessons
    AS individual,
    SUM(COUNT(*) FILTER (WHERE lessons_id = 2)) OVER ()        -- Number of group lessons
    AS group_lessons,
    SUM(COUNT(*) FILTER (WHERE lessons_id = 3)) OVER ()        -- Number of ensemble lessons
    AS ensemble,
    SUM(COUNT(*)) OVER () AS total                             -- Total number of lessons
FROM complet_lessons
WHERE EXTRACT(YEAR FROM period_start) = 2024                   -- Filter by year (2024)
GROUP BY TO_CHAR(period_start, 'Mon'), EXTRACT(MONTH FROM       -- Group by month
period_start)
ORDER BY EXTRACT(MONTH FROM period_start);                      -- Sort by month order
```

**Parsing the request:**

1. TO_CHAR(period_start, 'Mon'): This function extracts the month name in abbreviated format (e.g. "Jan", "Feb", "Mar").

2. COUNT(*) FILTER (WHERE lessons_id = 1): We use the FILTER construct to count the number of lessons of each type (individual, group, ensemble). We filter rows by each lesson type.

3. COUNT(*): The total number of lessons in each month. This is simply a count of all the rows (lessons) for each month.

4. WHERE EXTRACT(YEAR FROM period_start) = 2024: We filter the data by year (2024).

5. GROUP BY TO_CHAR(period_start, 'Mon'): We group the data by month.

6. ORDER BY EXTRACT(MONTH FROM period_start): Sort the results by month order (January to December).

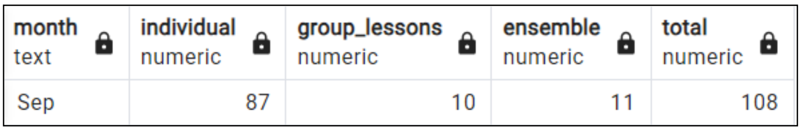
7. To calculate the totals (or grand totals) by column, we will use window functions to get the grand totals for all months.

However, it is important to remember that to calculate the grand total, we will need to aggregate across all rows, rather than group by month.

SUM(...) OVER (): This is a window function that calculates the grand total for the entire sample. In this case, it sums all lessons by type (individual, group, ensemble) and the total number of lessons. Using window functions allows us to calculate the grand totals at the row level, without requiring an additional UNION or subqueries.

**Query result:**

Since the complet_lessons table only has data for one month (September 2024), the query will return a table only for that month.

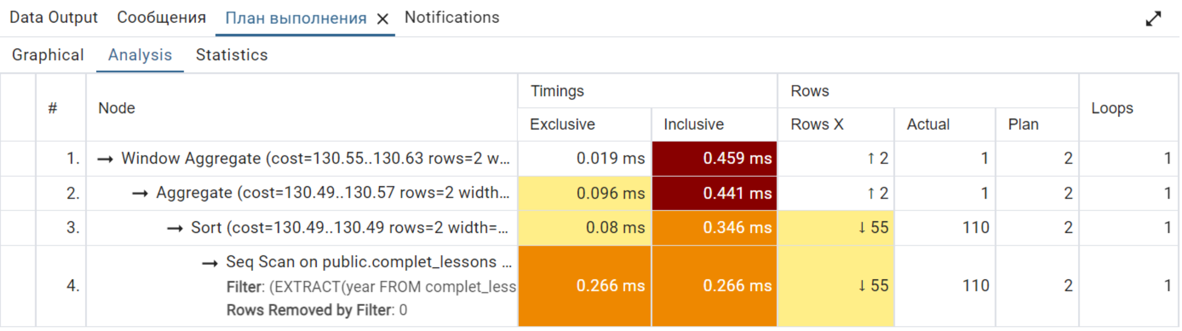


| month<br>text | individual<br>numeric | group_lessons<br>numeric | ensemble<br>numeric | total<br>numeric |
|---|---|---|---|---|
| Sep | 87 | 10 | 11 | 108 |

## Task 2:

**You need to analyze the performance of one of the queries using EXPLAIN ANALYZE.**

### Let's analyze the efficiency of query #1 using EXPLAIN ANALYZE:



| # | Node | Timings |  | Rows |  |  | Loops |
|---|---|---|---|---|---|---|---|
|  |  | Exclusive | Inclusive | Rows X | Actual | Plan |  |
| 1. | → Window Aggregate (cost=130.55..130.63 rows=2 w... | 0.019 ms | 0.459 ms | ↑ 2 | 1 | 2 | 1 |
| 2. | → Aggregate (cost=130.49..130.57 rows=2 width... | 0.096 ms | 0.441 ms | ↑ 2 | 1 | 2 | 1 |
| 3. | → Sort (cost=130.49..130.49 rows=2 width=... | 0.08 ms | 0.346 ms | ↓ 55 | 110 | 2 | 1 |
| 4. | → Seq Scan on public.complet_lessons ...<br>**Filter**: (EXTRACT(year FROM complet_less<br>**Rows Removed by Filter**: 0 | 0.266 ms | 0.266 ms | ↓ 55 | 110 | 2 | 1 |

This query plan shows the steps involved in executing a SQL query in PostgreSQL and contains important information about how the data is processed.

### Description of each stage of the plan:

1. **Window Aggregate:**

    - The stage where the window function is used to calculate the totals. This is the topmost node, indicating that the result is a windowed aggregation.
    - **Rows**: Indicates that the final result contains 2 rows.
    - **Timings**: Exclusive - the time spent only on this operation, Inclusive - the total time of the current operation and all operations below it.

2. **Aggregate:**

    - This step performs an aggregate calculation, such as COUNT or SUM. It generates results for grouping by month.
    - **Rows**: shows that 2 rows were generated here as well.

3. **Sort:**

    - The stage of sorting data by query criteria, such as month order. At this stage, PostgreSQL sorts all rows for proper output.
    - **Rows**: 110 rows sorted, which corresponds to the number of rows read from the original data.

4. **Seq Scan:**

    - This step shows that PostgreSQL performs a sequential scan of the entire complet_lessons table, applying the filter EXTRACT(YEAR FROM period_start) = 2024.
    - **Rows Removed by Filter**: 0, which means that all rows match the filter.

**Key points:**

- **Execution Speed:** This plan shows that the query executed very quickly (less than 1ms in most stages), indicating good performance.
- **Number of Rows:** The total number of rows in the complet_lessons table is 110. After applying window aggregation and grouping, there are only 2 rows left, as the result aggregates data by month.

**Conclusion:**

The query plan shows the sequential steps of data processing: from filtering rows by year, sorting and grouping to executing window functions. If optimization is required, you can consider using indexes on the period_start column to speed up filtering and sorting.

**CREATE INDEX idx_complet_lessons_period_start ON complet_lessons (period_start);**

**Request №2**

**Show how many students there are with no sibling, with one sibling, with two siblings, etc. This query is expected to be performed a few times per week. The database must contain students with no sibling, one sibling and two siblings, but doesn't have to contain students with more than two siblings. Note that it's not allowed to solve this by just adding a column with sibling count (maybe called no_of_siblings or something similar) to the student table. Such a solution would be almost impossible to maintain since it doesn't tell who's a sibling of who. If a student quits, there wont be any way to update the sibling count of that student's siblings. Table 2 below is an example result of such a query, illustrating the expected output.**

**Table 2. Expected output for query 2. This example is only meant to illustrate the expected rows and columns. It's perfectly fine to change text formatting, and also to change the values in the No of Students column. The values in the No of Siblings column must however be as specified: 0, 1, 2.**

| No of Siblings | No of Students |
|---|---|
| 0 | 3 |
| 1 | 2 |
| 2 | 5 |

To solve this problem, where it is necessary to count the number of students with a certain number of siblings, you can use a SQL query with aggregation based on the relationship between the students.

There is a student table and a sibling table that stores data about brothers and sisters.

**Table structure:**

- **student** — information about students.
- **student_numb** — unique identifier of the student.
- **sibling** — information about brothers and sisters.
- **student_numb** — identifier of the student.
- **person_sibling_id** — identifier of his or her brother or sister.

### Request:

```
SELECT
   no_of_siblings,                                                   -- Number of brothers and sisters
   COUNT(*) AS number_of_students                                    -- Number of students for a given number of
                                                                     siblings
FROM (
      SELECT
         s1.student_numb,
         COUNT(s2.person_sibling_id) AS no_of_siblings               -- Number of brothers and sisters
      FROM student s1
      LEFT JOIN sibling s2
         ON s1.student_numb = s2.student_numb OR
s2.person_sibling =s1.student_numb

     GROUP BY s1.student_numb

) AS sibling_counts

GROUP BY no_of_siblings
ORDER BY no_of_siblings;
```

### Parsing the request:

### Inner subquery:

We do a LEFT JOIN between the student and sibling tables to link each student to their siblings.

For each student (s1. student_numb), we count how many siblings they have. This is done using COUNT(s2. person_sibling_id) in the subquery.

We use a LEFT JOIN to include even those students who don't have siblings - in this case, their sibling count will be 0.

### External request:

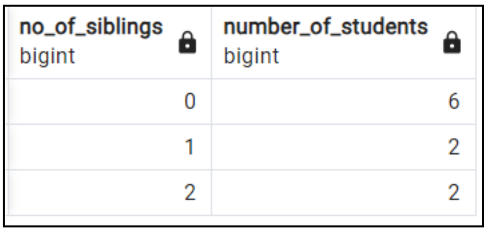
Using COUNT(*) we count the number of students for a given number of siblings, within the subquery the rows are unique by student_numb.

We group the results at the top level, by the number of siblings (no_of_siblings).

ORDER BY is at the top level, where it is applied to the results of the final query.

**Query result:**

In this example, 6 students have no siblings, 2 students have one sibling, and 2 students have two siblings.



| no_of_siblings<br>bigint | number_of_students<br>bigint |
|---:|---:|
| 0 | 6 |
| 1 | 2 |
| 2 | 2 |

**Request №3**

**List ids and names of all instructors who has given more than a specific number of lessons during the current month. Sum all lessons, independent of type, and sort the result by the number of given lessons. This query will be used to find instructors risking to work too much, and will be executed daily. Table 3 below is an example result of such a query, illustrating the expected output.**

**Table 3. Expected output for query 3. This example is only meant to illustrate the expected rows and columns. It's perfectly fine to change text formatting, and also to change the values.**

| Instructor Id | First Name | Last Name | No of Lessons |
|:---:|:---:|:---:|:---:|
| 13 | Albus | Dumbledore | 5 |
| 15 | Pomona | Sprout | 5 |
| 2 | Gilderoy | Lockhart | 4 |

To list the IDs and names of teachers who have taught more than a certain number of lessons in the current month, you can use a SQL query with aggregation. We will use the EXTRACT function to extract the current month from the lesson date, and filter by date, and count the number of lessons each teacher has taught in that month.

There are two tables:

instructor— information about instructors:

- instruktor_numb— unique instructor identifier.
- name_inst— instructor's first name.
- surname_inst— instructor's last name.

complete_lessons— information about completed lessons:

- lessons_id— unique lesson identifier.
- instructor_id— instructor identifier (foreign key linking to the instructors table).
- period_start— date of the lesson.

**Request:**

```
SELECT
  i.instruktor_numb,
  i.name_inst,
  i.surname_inst,
  COUNT(a.lessons_id) AS lesson_count
FROM instructor i
LEFT JOIN complet_lessons a ON i.instruktor_numb = a.instructor_id
AND EXTRACT(MONTH FROM a.period_start) = EXTRACT(MONTH FROM          -- Filter by current month
CURRENT_DATE)
AND EXTRACT(YEAR FROM a.period_start) = EXTRACT(YEAR FROM CURRENT_DATE)   -- Filter by current year
GROUP BY i.instruktor_numb, i.name_inst, i.surname_inst
HAVING COUNT(a.lessons_id) > 3                                       -- Threshold: more than 3
                                                                     lessons
ORDER BY lesson_count DESC;                                          -- Sort by number of
                                                                     lessons
```

**Parsing the request:**

1. EXTRACT(MONTH FROM a.period_start) = EXTRACT(MONTH FROM CURRENT_DATE) This part of the query extracts the month from the lesson date (a.period_start) and compares it to the current month. This allows us to select only those lessons that were taught in the current month.

2. EXTRACT(YEAR FROM a.period_start) = EXTRACT(YEAR FROM CURRENT_DATE): Here we add filtering by the current year to avoid selecting lessons from the previous year.
3. JOIN complet_lessons a ON i.instruktor_numb = a.instructor_id We join the instructor table to the lessons table on the teacher_id field to get all the lessons taught by each instructor.
4. GROUP BY i.instruktor_numb, i.name_inst, i.surname_inst We group the data by instructors to count the number of lessons for each instructor.
5. HAVING COUNT(a.lessons_id) > 3: We select only those instructors who have taught more than 3 lessons in the current month. This condition can be changed to any other value (for example, HAVING COUNT(a.lessons_id) > 5).
6. ORDER BY lesson_count DESC: The results are sorted by the number of lessons taught in descending order, so that the instructors with the most lessons are first.

**Query result:**

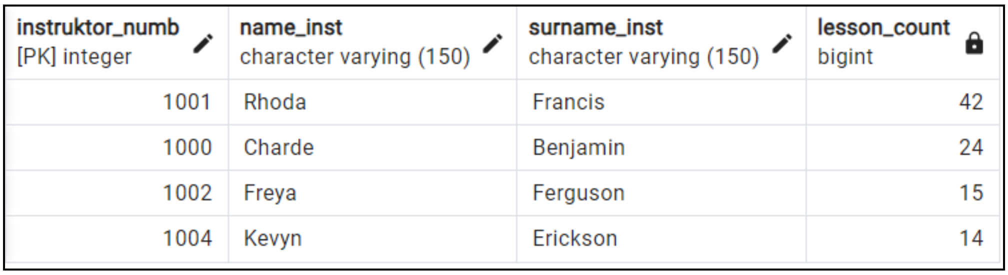
Since the data in the complet_lessons table is loaded only for September 2024, we modify the query as follows:

```
AND EXTRACT(MONTH FROM a.period_start) = 9          -- Filtering data by September
 AND EXTRACT(YEAR FROM a.period_start) = 2024       -- Filter by 2024
```

Number of Lessons - The number of lessons taught by the teacher in the current month, if it is greater than the specified threshold.

You can filter by a different number of lessons by simply changing the value in the HAVING COUNT(a.lessons_id) > 3 condition.

This query can be run daily, and for each day it will display teachers who have taught more than the specified number of lessons in the current month.



| instruktor_numb<br>[PK] integer | name_inst<br>character varying (150) | surname_inst<br>character varying (150) | lesson_count<br>bigint |
|---|---|---|---|
| 1001 | Rhoda | Francis | 42 |
| 1000 | Charde | Benjamin | 24 |
| 1002 | Freya | Ferguson | 15 |
| 1004 | Kevyn | Erickson | 14 |

**Request № 4**

**List all ensembles held during the next week, sorted by music genre and weekday. For each ensemble tell whether it's full booked, has 1-2 seats left or has more seats left. HINT: you might want to use a CASE statement in your query to produce the desired output. Table 4 below is an example result of such a query, illustrating the expected output.**

**Table 4. Expected output for query 4. This example is only meant to illustrate the expected rows and columns. It's perfectly fine to change text formatting, and also to change the values.**

| **Day** | **Genre** | **No of Free Seats** |
|---|---|---|
| Tue | Gospel | No Seats |
| Wed | Punk | 1 or 2 Seats |
| Wed | Rock | No Seats |
| Fri | Rock | Many Seats |

To execute a query that lists all the bands that will be playing next week, sorted by musical genre and day of the week, we will use the following approach:

1. Filter by dates - we need to extract dates for the next week. In PostgreSQL, we can use date functions such as CURRENT_DATE and INTERVAL for this.
2. Count available seats - for each ensemble, we need to calculate the number of seats left and use a CASE construct to display how many seats are left, given the specified conditions (e.g. "many seats", "1 or 2 seats", "no seats").
3. Sort by genre and day of the week - after performing the aggregation and counting the number of seats, the results should be sorted by musical genre and day of the week.

There are the following tables:

1. **ensemble** — information about ensembles:

   - ensem_id — ensemble identifier.
   - gentre_ensembles — musical genre of the ensemble.
   - data_ensem — date and time of the ensemble.
   - max_students — maximum number of seats in the ensemble.

2. **actual_seats_ensemble** — information about the actual number of ensemble members:
    - reserv_id — ensemble identifier.
    - reserved_seats — number of actually booked seats.

Request**:**

```
SELECT
TO_CHAR(e.data_ensem, 'Day') AS day_of_week,                   -- Day of the week
g.gentre_ensembles AS genre,                                   -- Musical genre
CASE
WHEN (e.max_students - COALESCE(SUM(r.reserved_seats), 0)) =    -- If there are no vacancies
0 THEN ' No seats available '
WHEN (e.max_students - COALESCE(SUM(r.reserved_seats), 0)) <=   -- If there are 1 or 2 places left
2 THEN ' 1 or 2 places '
    ELSE ' Lots of places '                                    -- If there are more than 2 places left
END AS available_seats
FROM ensemble e
LEFT JOIN actual_seats_ensemble r ON e.ensem_id = r.reserv_id
LEFT JOIN ensemble_genre g ON e.ensembles_id = g.ensembles_id
WHERE e.data_ensem <= CURRENT_DATE + INTERVAL '1 day'          -- Beginning of next week
AND e.data_ensem < CURRENT_DATE + INTERVAL '8 days'            -- End of next week
GROUP BY e.ensem_id, g.gentre_ensembles, e.data_ensem
ORDER BY g.gentre_ensembles, TO_CHAR(e. data_ensem, 'D');      -- Sort by genre and day of the week
```

**Parsing the request**:

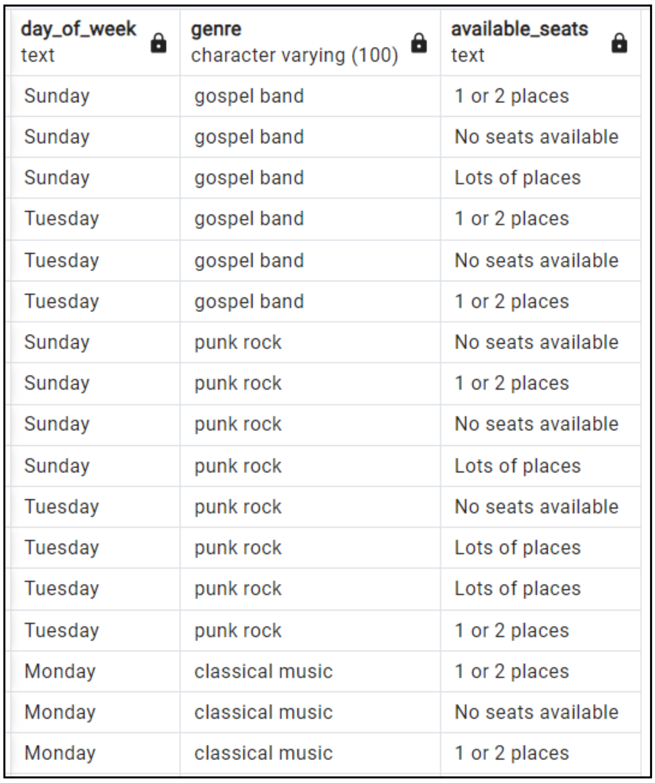
1. TO_CHAR(e.ensemble_date, 'Day'): We extract the day of the week for each ensemble to display in a human-readable format, such as "Monday", "Tuesday", etc. It is important to note that in PostgreSQL the day of the week may be displayed with spaces by default, so we can use the TRIM function to remove the extra spaces.
2. Using LEFT JOIN between ensembles and actual_seats_ensemble tables: We join the ensembles table with the reservations table to count the number of actual seats for each ensemble. This allows us to account for ensembles with no reservations (for which the number of actual seats booked will be 0).

3. Using COALESCE(SUM(r.reserved_seats), 0): This is necessary so that in case of no reservations (when there are no records in the actual_seats_ensemble table for the ensemble) the result of the sum will be 0. COALESCE replaces NULL with 0**.**
4. CASE condition: We calculate how many seats are left:

    - If there are no free seats, we output "No seats".
    - If there are 1 or 2 seats left, we output "1 or 2 seats".
    - If there are more than 2 seats, we output "Many seats".

5. Filtering by dates: We filter for ensembles that take place in the next week. To do this, we use CURRENT_DATE + INTERVAL '1 day' for the start of the next week and CURRENT_DATE + INTERVAL '8 days' for the end of the next week (to capture all ensembles that take place from Monday to Sunday).
6. Grouping: We group the results by ensemble, music genre, and ensemble date to calculate the number of reservations for each ensemble.
7. Sorting: We sort the results first by music genre, then by day of the week (we use TO_CHAR(e.ensemble_date, 'D') to do this, which returns the numeric value of the day of the week, from 1 to 7).

**Query result:**



| day_of_week<br>text | genre<br>character varying (100) | available_seats<br>text |
|---|---|---|
| Sunday | gospel band | 1 or 2 places |
| Sunday | gospel band | No seats available |
| Sunday | gospel band | Lots of places |
| Tuesday | gospel band | 1 or 2 places |
| Tuesday | gospel band | No seats available |
| Tuesday | gospel band | 1 or 2 places |
| Sunday | punk rock | No seats available |
| Sunday | punk rock | 1 or 2 places |
| Sunday | punk rock | No seats available |
| Sunday | punk rock | Lots of places |
| Tuesday | punk rock | No seats available |
| Tuesday | punk rock | Lots of places |
| Tuesday | punk rock | Lots of places |
| Tuesday | punk rock | 1 or 2 places |
| Monday | classical music | 1 or 2 places |
| Monday | classical music | No seats available |
| Monday | classical music | 1 or 2 places |

# 5 Discussion

**1. Are views and materialized views used in all queries that benefit from using them? Can any query be made easier to understand by storing part of it in a view? Can performance be improved by using a materialized view?**

Using views and materialized views can make queries much easier and faster, but not all queries benefit from them. To understand when to use them, it's important to understand what they are and what problems they can solve.

### 1. Views (examples of views are posted on github, file VIEW_.sql in /KTX-1167/seminar3/)

**Views** are stored SQL queries that can be used to simplify working with a database. They do not contain any data, but merely store the structure of a query. When a view is called, the SQL query associated with that view is executed.

**Benefits of Views:**

- **Simplify complex queries**: Views can hide the complexity of SQL queries by joining multiple tables or performing aggregations. This makes queries simpler and easier to understand for users who do not want to work with long or complex SQL queries.
- **Reusability**: A query written once as a view can be reused in different parts of the application or in other queries.
- **Security**: Views can be used to provide limited access to data. For example, a view can exclude certain fields or aggregate data, hiding the complex internal structures of the database**.**

**Disadvantages of Views:**

- **Performance**: Views do not store results on disk, and each time a view is called, the associated SQL query is executed. This can be slow, especially if the query is complex or works with large amounts of data.
- **Loss of flexibility**: Since a view is defined by a SQL query, it cannot be easily modified for specific needs. If the query requires additional parameters or changes, it will require re-creating the view**.**

### When to Use Views:

Views are useful when:

- Queries are repetitive and need to be simplified.
- Complex operations need to be hidden from end users.
- Restrict access to data or presentation of data in a specific format.

### 2. Materialized Views

**Materialized views** are views whose results are stored on disk. This means that the data used in the query is pre-computed and stored in a physical table. Materialized views require explicit refresh when data changes in the source tables.

### Benefits of Materialized Views:

- **Performance:** Since the data is already computed and stored on disk, queries against materialized views are faster than if they were to perform the calculations on the fly. This is especially useful for complex and resource-intensive queries such as aggregations and joins.
- **Reduced database load**: In the case of complex queries that require multiple calculations, using materialized views helps avoid repetitive operations by storing the results on disk**.**

### Disadvantages of Materialized Views:

- **Out-of-date data**: Materialized views are not automatically refreshed with every change to the underlying tables. To get current data, you must manually refresh the materialized view using the REFRESH MATERIALIZED VIEW command.
- **Use of additional disk space**: Because results are persisted, materialized views require additional storage space.

### When to Use Materialized Views:

Materialized views are useful when:

- Queries are executed frequently, but data changes infrequently.

- Need to speed up resource-intensive queries, such as aggregations or joins on large tables.
- Priority is given to performance, not always-up-to-date data.
- Materialized views can speed up complex queries by storing pre-computed data, but require additional space and time to update the data.

Both tools can improve the performance and usability of the database, but their use should be conscious and appropriate for those queries that really benefit from using these mechanisms.

**2. Did you change the database design to simplify these queries? If so, was the database design worsened in any way just to make it easier to write these particular queries?**

The database structure has not been changed in any way. The database structure developed is ideal for writing these specific queries.

**3. Is there any correlated subquery, that is a subquery using values from the outer query? Remember that correlated subqueries are slow since they are evaluated once for each row processed in the outer query.**

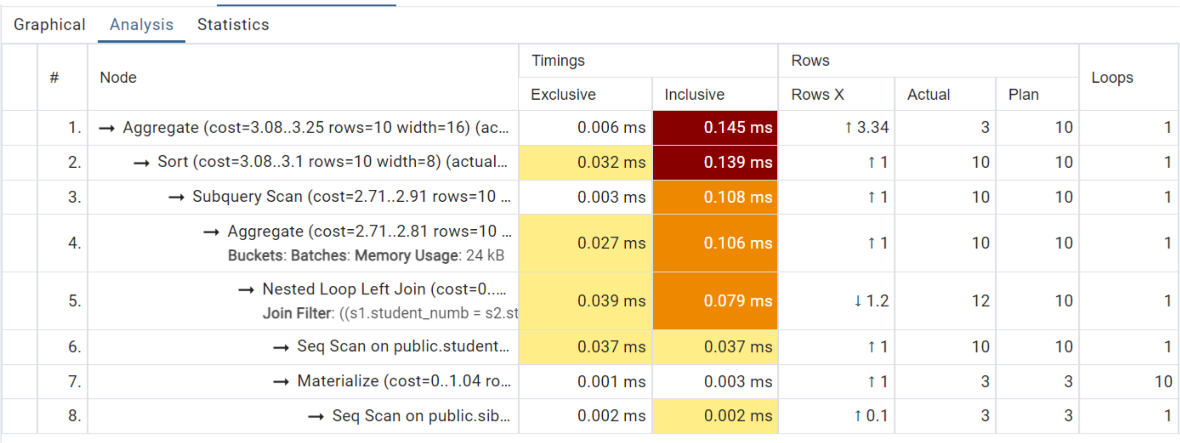
Yes, there is. Request №2 has a correlated subquery, which is a subquery that uses values from the outer query. Yes, a correlated subquery is slower because they are evaluated once for each row processed in the outer query.

**4. Are there unnecessarily long and complicated queries? Are you for example using a UNION clause where it's not required?**

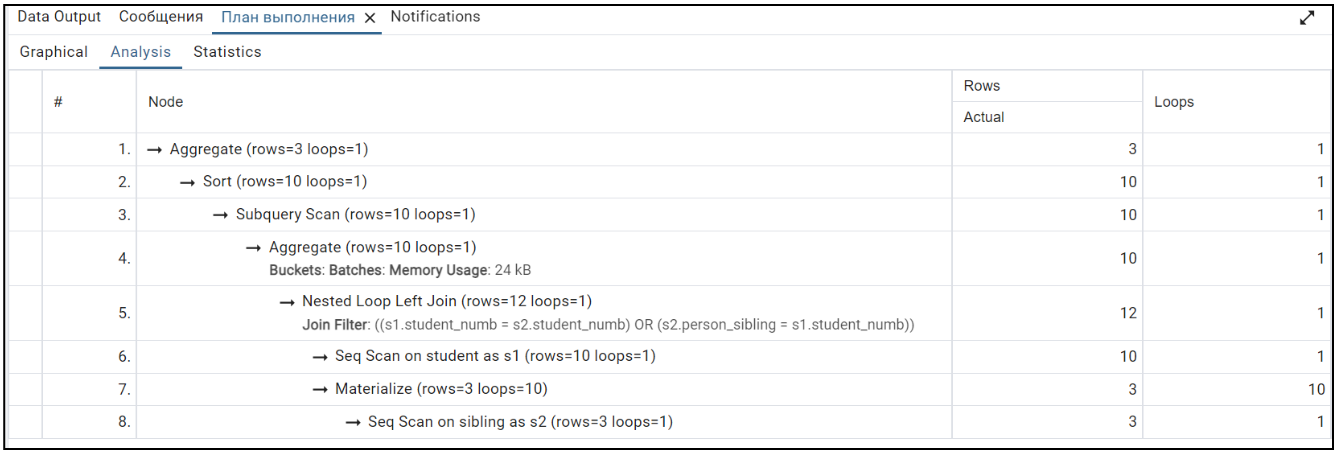
No, there are no unnecessarily long and complex queries. The UNION clause is not used where it is not required.

**5. Analyze the query plan for at least one of your queries using the command EXPLAIN (or EXPLAIN ANALYZE), which is available in both Postgres and MySQL. Where in the query does the DBMS spend most time? Is that reasonable? If you have time, also consider if the query can be rewritten to execute faster, but you're not required to do that. The postgres documentation is found at https://www.postgresql.org/docs/current/using-explain.html and https://www.postgresql.org/docs/current/sql-explain.html. There's also some explanation of EXPLAIN in the document Tips and Tricks for Project Task 3.**

**Let's analyze Request #2.** The request was executed successfully. Total execution time: 218 msec. rows processed: 1.



| # | Node | Timings | | Rows | | | Loops |
|---|---|---|---|---|---|---|---|
| | | Exclusive | Inclusive | Rows X | Actual | Plan | |
| 1. | → Aggregate (cost=3.08..3.25 rows=10 width=16) (ac... | 0.006 ms | 0.145 ms | ↑ 3.34 | 3 | 10 | 1 |
| 2. | → Sort (cost=3.08..3.1 rows=10 width=8) (actual... | 0.032 ms | 0.139 ms | ↑ 1 | 10 | 10 | 1 |
| 3. | → Subquery Scan (cost=2.71..2.91 rows=10 ... | 0.003 ms | 0.108 ms | ↑ 1 | 10 | 10 | 1 |
| 4. | → Aggregate (cost=2.71..2.81 rows=10 ...<br>**Buckets**: **Batches**: **Memory Usage**: 24 kB | 0.027 ms | 0.106 ms | ↑ 1 | 10 | 10 | 1 |
| 5. | → Nested Loop Left Join (cost=0.....<br>**Join Filter**: ((s1.student_numb = s2.st | 0.039 ms | 0.079 ms | ↓ 1.2 | 12 | 10 | 1 |
| 6. | → Seq Scan on public.student... | 0.037 ms | 0.037 ms | ↑ 1 | 10 | 10 | 1 |
| 7. | → Materialize (cost=0..1.04 ro... | 0.001 ms | 0.003 ms | ↑ 1 | 3 | 3 | 10 |
| 8. | → Seq Scan on public.sib... | 0.002 ms | 0.002 ms | ↑ 0.1 | 3 | 3 | 1 |



| # | Node | Rows | Loops |
|---|---|---|---|
| | | Actual | |
| 1. | → Aggregate (rows=3 loops=1) | 3 | 1 |
| 2. | → Sort (rows=10 loops=1) | 10 | 1 |
| 3. | → Subquery Scan (rows=10 loops=1) | 10 | 1 |
| 4. | → Aggregate (rows=10 loops=1)<br>Buckets: Batches: Memory Usage: 24 kB | 10 | 1 |
| 5. | → Nested Loop Left Join (rows=12 loops=1)<br>Join Filter: ((s1.student_numb = s2.student_numb) OR (s2.person_sibling = s1.student_numb)) | 12 | 1 |
| 6. | → Seq Scan on student as s1 (rows=10 loops=1) | 10 | 1 |
| 7. | → Materialize (rows=3 loops=10) | 3 | 10 |
| 8. | → Seq Scan on sibling as s2 (rows=3 loops=1) | 3 | 1 |

**The general gist of the plan is:**

**Stage 1:** PostgreSQL performs a sequential scan of the student (10 rows) and sibling (3 rows) tables.

**Stage 2:** For each row from the student table, PostgreSQL joins it to the sibling table using the join condition (s1.student_numb = s2.student_numb) OR (s2.person_sibling = s1.student_numb). This is done using nested loops (Nested Loop Left Join).

**Stage 3:** The results of the join are subject to aggregate processing (e.g. row count or other grouping).

**Stage 4:** The intermediate data is sorted and/or materialized for optimization.

**To optimize the query, we will use indexes to speed up join operations (Nested Loop).** Indexes allow PostgreSQL to avoid full table scans (Seq Scan) and use indexes (Index Scan) for faster row access.

```
CREATE INDEX idx_student_numb ON student(student_numb);
CREATE INDEX idx_sibling_student_numb ON sibling(student_numb);
CREATE INDEX idx_sibling_person_sibling ON sibling(person_sibling);
```

## Higher Grade Part

**The task is that, for marketing purposes, the Soundgood music school wants to be able to see which lessons each student has taken, and at which cost, since they first joined the school. This means you are required to keep records of all given lessons, including participants and price for each lesson. One way to do this would be to just save a copy of the entire database each day, or each time a price is changed. This is however unnecessarily complicated, since it will be slow to generate reports from such a database, and since you would have to add some kind of time interval for the prices, instead of just keeping the current price. A better way to solve the problem is to create a denormalized historical database. One example of an appropriate denormalization is to store prices of lessons in a column in the lessons table, instead of keeping prices in a separate table, even though this will lead to duplicated data, with the same price appearing in all lessons of the same type. Another possible denormalization is to merge individual, group and ensemble lessons into one single lesson table.**

**Create a historical database like the one described above, and also create SQL statements for copying data from your present database to the historical database. The historical database shall contain the following data: lesson type (group, individual or ensemble), genre (empty if it's not en ensemble), instrument (empty if it's an ensemble), lesson price, student name, student email. You're not required to write a program that automates moving data to the historical database, it's sufficient to create SQL statements that are executed manually. Discuss advantages and disadvantages of using denormalization. This discussion shall be placed in the Discussion chapter of the report.**

## Solution to the problem

To create a denormalized historical database for Soundgood Music School, we need to develop a table structure and SQL statements that will allow us to copy data from the current database to the historical one. Let's go through this step by step.

### 1. Structure of the historical database

To store all the data about the lessons conducted, including their price, type, genre, instrument and student information, we can create the following table.

| | |
|---|---|
| CREATE TABLE historical_lessons ( | |
| lesson_id SERIAL PRIMARY KEY, | -- Unique lesson identifier |
| lesson_date DATE NOT NULL, | -- Start date of the lesson |
| lesson_type VARCHAR(20) NOT NULL, | -- Type of lesson: individual, group or ensemble |
| genre VARCHAR(50), | -- Genre (empty if it's not en ensemble) |
| instrument VARCHAR(50), | -- Instrument (empty if it's an ensemble) |
| lesson_price DECIMAL(10, 2) NOT NULL, | -- Price of the lesson |
| student_firstname VARCHAR(100) NOT NULL, | -- Student's firstname |
| student_surname VARCHAR(100) NOT NULL, | -- Student's surname |
| student_email VARCHAR(100) NOT NULL<br>); | -- Student's email |

## 2. SQL statements for copying data

Now let's create SQL queries to copy data from the current database to this denormalized table.

We have the following tables in the current database:

**booking_individual** (individual lessons):
lesson_id, period_start, lesson_price, instrument_id, student_numb, lessons_id, level_id
**group_lessons** (group lessons):
lesson_id, period_start, lesson_price, student_numb, lessons_id
**ensemble** (ensemble lessons):
lesson_id, period_start, lesson_price, instrument_id, student_numb, lessons_id
**students**:
student_id, student_numb,
**contact_detail:**
student_email
**price:**
price_service

Now let's write SQL queries to copy data to the history table.

### Copying data from individual lessons:

```
INSERT INTO historical_lessons (lesson_date, lesson_type, genre, instrument, lesson_price,
student_firstname, student_surname, student_email)


SELECT
    i.period_start,
    'Individual' AS lesson_type,
    'No genre' AS genre,
    m.instruments,
    p.price_service,
    s.firstname,
    s.surname,
    c.e_mail
FROM booking_individual i
LEFT JOIN student s
    ON i.student_numb = s.student_numb
LEFT JOIN contact_detail c
    ON s.details_id = c.contact_id
LEFT JOIN mus_instruments m
    ON i.instrument_id = m.instrument_id
LEFT JOIN price p
    ON i.lessons_id = p.lessons_id
    AND i.level_id = p.levels_id
WHERE i.lessons_id = 1
  AND i.level_id IN (1, 2, 3);
```

### Copying data from group lessons:

```
INSERT INTO historical_lessons (lesson_date, lesson_type, genre, instrument, lesson_price,
student_firstname, student_surname, student_email)


SELECT
    g.data_group,
    'Group' AS lesson_type,
    'No genre' AS Genre,
    m.instruments,
      p.price_service,
      s.firstname,
      s.surname,
      c.e_mail
   FROM group_lessons g
   LEFT JOIN student s
      ON g.students_numb = s.student_numb
   LEFT JOIN contact_detail c
      ON s.details_id = c.contact_id
   LEFT JOIN mus_instruments m
      ON g.instrument_id = m.instrument_id
   LEFT JOIN price p
      ON g.lessons_id = p.lessons_id
   WHERE g.lessons_id = 2;
```

Copying data from ensemble lessons:

```
   INSERT INTO historical_lessons (lesson_date, lesson_type, genre, instrument, lesson_price,
student_firstname, student_surname, student_email)

   SELECT
      e.data_ensem,
      'Ensemble' AS lesson_type,
      gg.gentre_ensembles AS genre,
      'Different musical instruments' AS instrument,
      p.price_service,
      s.firstname,
      s.surname,
      c.e_mail
   FROM ensemble e
   LEFT JOIN student s
      ON e.students_numb = s.student_numb
   LEFT JOIN contact_detail c
      ON s.details_id = c.contact_id
   LEFT JOIN ensemble_genre gg
```

```
    ON e.ensembles_id = gg.ensembles_id
LEFT JOIN price p
    ON e.lessons_id = p.lessons_id
    WHERE e.lessons_id = 3;
```

**Explanation of operators:**

For each lesson type, we select the columns we want and fill in the fields in the history table.

We use a LEFT JOIN to join the students table and each lesson table to get the student's name and email.

For ensemble lessons, we add additional information about the genre and instrument that is not present for individual and group lessons.

**3. Advantages and Disadvantages of Denormalization**

**Benefits of Denormalization**:

**Speed up data reading**: Denormalized tables are often used for analytical queries because they contain all the necessary data in one place. This reduces the number of table joins (JOINs), which speeds up query execution.

**Simplifying queries**: Denormalized data allows you to reduce the complexity of SQL queries. For example, to get all the information about the lessons taught (including student, genre, and price data), you don't need to perform several JOIN queries — all the information is already contained in one table.

**Data historiation**: All changes to lesson data are recorded in a historical table. This is important for analyzing student and lesson data over time, as well as for marketing and reporting.

**Disadvantages of denormalization:**

Data Redundancy: Denormalization results in duplication of information. For example, if the same student attends multiple classes, their name and email will be recorded for each class, increasing the size of the database and potentially wasting memory.

Issues with updating data: When data in the source tables changes (for example, changing a student's name), the data must be updated in all places where it is used. Unlike normalized tables, where data is updated in one place, denormalized tables may require updates to be performed on multiple rows.

Data Maintenance Challenges: If any information changes (e.g. changes in lesson price), this will require additional logic to maintain data integrity, as updates may require changes to many rows in the history table..

**Conclusion**

Denormalization can be useful for reporting and data analysis, where query speed and ease of retrieval are important. However, it may be less effective for update operations, as it requires more effort to maintain data integrity. Therefore, the decision to denormalize should be made based on the performance and data maintenance requirements of your system.

# 6 Comments About the Course

The entire time after the second seminar was spent on solving the SQL assignment, including lectures, describing and explaining the basic concepts, principles and theory in the field of data/databases/data warehousing, as well as in the field of information administration and database design, creating queries and reports for business analysis, analyzing the efficiency of queries and preparing for the seminar. I believe that solving such a large practical task will help me a lot in the future.