Seminar 3 - SQL

Data Storage Paradigms, IV1351

Seema Bashir

4/12-2023

1 Introduction

The seminar's objective is to interact with the SoundGood Music School database utilizing SQL, the Structured Query Language. The primary focus is on creating Online Analytical Processing (OLAP) queries and views to facilitate comprehensive business analysis and reporting. In addition, four queries are manually generated and executed to produce analysis reports. The evaluation of a chosen query's efficiency is conducted through EXPLAIN ANALYZE, a tool present in both PostgreSQL and MySQL. The overarching goal is a comprehensive exploration and optimization of SQL's analytical capabilities within the context of the SoundGood Music School.

In collaboration with Razan Yakoub, this report details the approach employed and executed, providing an overview of the collaborative results in meeting the specified requirements.

2 Literature Study

In order to prepare for this seminar, chapters 6 and 7 in the 7th edition of Fundamentals of Database Systems by Elmasri and Navathe are evaluated.

Furthermore, a lecture on SQL provided by Paris Carbone is watched and evaluated in order to gain a better understanding of the domain-specific language (SQL) used for managing and manipulating relational databases.The content extensively covers fundamental topics related to relational databases, declarative query languages, and the Structured Query Language (SQL). It provides a comprehensive understanding of key concepts such as tuple relational calculus, domain relational calculus, and SQL syntax. The material also addresses critical data manipulation operations like insertions, deletions, and updates, in addition to exploring aggregation functions and the generation of views. It emphasizes the vital role of SQL in the field of data management, illustrating its significance through practical examples of SQL queries designed for various scenarios.

Lastly, the "Tips and Tricks 3" document provided vital details regarding the different functionalities which can be utilized to write advance queries. A vital point discussed in the document adheres to the process of of reading data spread across multiple tables in a database using SQL queries. The common scenario involves using JOIN clauses to merge tables based on a shared column. This clause also emphasizes the distinction between LEFT JOIN and FULL JOIN, depending on the existence of a common column. This clause also highlights extracting relevant data from the merged tables, selecting specific columns of interest, and filtering based on the desired department. The use of the CONCAT function to unify department numbers from different tables is introduced.

Additionlly, another point of significance in the document is the utlization of sub queries. The example underscores the importance of employing subqueries in SQL queries for enhanced flexibility and adaptability. By using subqueries, the query writer can replace static criteria, such as primary key values, with dynamic conditions, offering a more versatile approach. The use of subqueries adds a layer of abstraction to the query, promoting readability and maintainability. It enables SQL queries to be written in a more intuitive and expressive manner, aligning with real-world scenarios where conditions might be based on dynamic or contextual factors.

The fifth clause underscores the strategic importance of leveraging views and materialized views in SQL queries for efficient database operations. Non-materialized views are highlighted as valuable tools for code reuse, aiding in the avoidance of redundant queries and providing a means to name and explain complex SQL operations, akin to the organizational benefits of private methods in object-oriented programming. The example illustrates encapsulating a complex query within a view for reuse with different search criteria. Materialized views are introduced as a performance optimization, emphasizing their ability to store query results on disk for faster reading. However, the trade-off is noted, as materialized views require explicit refreshes when underlying tables are updated, prompting careful consideration based on the specific needs of read and write performance. The clause advocates for measuring these factors to inform decisions about whether to use materialized views, thus emphasizing a practical and strategic approach to optimizing SQL queries in database management.

Moreover, the document also emphasises on the analysis of queires. The section of the document focuses on the practical utilization of the EXPLAIN command in PostgreSQL for query analysis. It highlights EXPLAIN's role in revealing the intricacies of query execution plans and understanding the factors contributing to execution time. Acknowledging PostgreSQL's specificity, the section directs readers to additional documentation for a comprehensive understanding. The analysis emphasizes the importance of costs, rows, and width parameters at each node, providing a step-by-step breakdown of the execution plan.

3 Method

The database is implemented in the PostgresSQL database management system, and query development is facilitated through the pgAdmin graphical user interface tool. Data for the database is generated using an online tool, https://generatedata.com/. To ensure query accuracy, a comprehensive review of the entire database content is performed, requiring manual intervention. This process includes verifying the existence sufficient number of tuples in each table. Subsequently, the results of all queries are meticulously cross-checked with the current state of the database to confirm their accuracy. All queries are encapsulated within views to enhance readability.

It should be noted that before executing the main queries, a series of pre-queries is employed to evaluate the expected results and assess their validity. These preliminary queries serve as a proactive measure to anticipate and identify any potential issues or discrepancies in the data. This step involves verifying the appropriateness of the input parameters, ensuring the existence of necessary data elements, and confirming that the anticipated outcomes align with the objectives of the subsequent main queries.

Lastly, PostgreSQL's fdw module is utlilized for the creation of a distinct historical database for the higher-grade task. This module makes it easier for databases to connect to one another, facilitating easier communication and access to outside data sources. This feature is essential to the task at hand as it makes data management between the main SoundGood Music School database and the recently constructed historical database run smoother.

Direct access to SoundGood's data is made possible in this configuration by importing SoundGood Music School tables as foreign tables into the historical database. After that, fresh de-normalized tables within the historical database are made specifically for lesson and student data.

4 Results

The Git repository contains SQL files responsible for tasks such as database creation, data population, definition of views and queries, and the establishment of the historical database: `https://github.com/Seemamian/Data_Lagring.git`

4.1 Query 1: The total number of lessons per month during a specified year

```sql
/*Query 1*/
-- View To provide lesson statistics per month during a specifik year
CREATE VIEW LessonStatistics AS
SELECT
    EXTRACT(YEAR FROM l.date) AS year,
    EXTRACT(MONTH FROM l.date) AS month,
    COUNT(l.lesson_id) AS total_lessons,
    SUM(CASE WHEN lt.type = 'Individual' THEN 1 ELSE 0 END) AS individual_lessons,
    SUM(CASE WHEN lt.type = 'Group' THEN 1 ELSE 0 END) AS group_lessons,
    SUM(CASE WHEN lt.type = 'Ensemble' THEN 1 ELSE 0 END) AS ensemble_lessons
FROM
    lesson l
LEFT JOIN
    pricing_scheme ps ON l.price_id = ps.price_id
LEFT JOIN
    lesson_type lt ON ps.lessontype_id = lt.lessontype_id
GROUP BY
    EXTRACT(YEAR FROM l.date), EXTRACT(MONTH FROM l.date);

-- The final query
SELECT * FROM LessonStatistics
WHERE year = EXTRACT(YEAR FROM CURRENT_DATE)
ORDER BY
    year, month;
```

year numeric	month numeric	total_lessons bigint	individual_lessons bigint	group_lessons bigint	ensemble_lessons bigint
2023	1	4	2	0	2
2023	2	2	1	1	0
2023	3	6	3	1	2
2023	4	3	0	1	2
2023	5	3	0	2	1
2023	6	3	1	2	0
2023	7	8	3	2	3
2023	8	3	1	2	0
2023	9	5	1	3	1
2023	10	8	3	0	5
2023	11	2	1	0	1
2023	12	22	0	1	21

Figure 1: Code for Query 1 and the Resulting Table

The SQL query provided above is tailored to extract and organize crucial statistics related to lessons conducted over a specified year. The objective is to facilitate regular tracking of lesson distribution on a monthly basis. Inside the query, a view is first created named LessonStatistics.

The LessonStatistics view encapsulates a set of essential functions and logic designed to efficiently organize and present key statistics about lessons conducted over a specified period. The use of the EXTRACT function allows for the extraction of the year and month from the lesson dates, enabling a meaningful temporal grouping of the data. The subsequent functions, including COUNT and conditional SUM operations, provide valuable insights into the total number of lessons, as well as the distribution of lessons across different types—individual, group, and ensemble.

The final query exemplifies the practical application of the view, focusing on usability for regular analysis. By selecting relevant columns from LessonStatistics and filtering results based on the current year, the query allows users to obtain a concise summary of lesson statistics for each month. The distinct rows for total lessons, individual lessons, group lessons, and ensemble lessons ensure clarity, meeting the specified requirement that allows for separate rows for each category as long as the month is clearly identified.

This query is intended for frequent use, offering a streamlined and efficient means of extracting essential information regarding lesson distribution throughout the specified year. Its design aligns with the reporting needs of the user, allowing for a seamless integration into routine analysis and reporting processes.

4.2 Query 2: Sibling Count

```sql
/*Query 2*/
-- View for counting siblings
CREATE VIEW SiblingCounts AS
SELECT
    student_id,
    COUNT(*) AS no_of_siblings
FROM
    sibling_student
GROUP BY
    student_id;
-- View for getting distinct students
CREATE VIEW DistinctStudents AS
SELECT
    s.student_id
FROM
    student s
LEFT JOIN
    sibling_student ss ON s.student_id = ss.student_id
GROUP BY
    s.student_id;
--The final query
SELECT
    CASE
        WHEN siblings.no_of_siblings IS NULL THEN 0
        ELSE siblings.no_of_siblings
    END AS "No of Siblings",
    COUNT(students.student_id) AS "No of Students"
FROM
    SiblingCounts siblings
RIGHT JOIN
    DistinctStudents students ON siblings.student_id = students.student_id
WHERE
    (siblings.no_of_siblings IS NULL OR siblings.no_of_siblings <= 2)
GROUP BY
    siblings.no_of_siblings
ORDER BY
    "No of Siblings";
```

No of Siblings bigint	No of Students bigint
0	54
1	18
2	7

Figure 2: Code for Query 2 and the Resulting Table

Query 2 is devised to conduct an in-depth examination of student demographics, specifically delving into the distribution of students based on their respective sibling counts.The creation of distinct views, 'SiblingCounts' and 'DistinctStudents,' in the provided SQL code serves a deliberate and strategic purpose in enhancing the clarity and flexibility of the analysis. In the 'SiblingCounts' view, the count of siblings for each student is meticulously computed through a grouping of records in the 'sibling_student' table based on unique student ID's. On the other hand, the 'DistinctStudents' view focuses on compiling a comprehensive list of unique student identifiers, considering potential sibling relationships in the left join operation between the 'student' and 'sibling_student' tables. This separation of views optimizes modularity and facilitates code reuse, enabling users to independently query and analyze specific aspects of the dataset.

Thereby, the two view: SiblingCounts' and 'DistinctStudents,' are employed. They are used to analyze student data without directly modifying the original 'student' table. 'SiblingCounts' calculates the number of siblings for each student, while 'DistinctStudents' ensures a list of unique student identifiers is considered.The main query then uses these views to discern the distribution of students based on sibling counts. The 'CASE' statement handles NULL values for sibling counts, substituting them with 0. The 'RIGHT JOIN' ensures all students, including those without siblings, are included. The query filters results to include only students with no siblings or up to two siblings. This approach allows for a detailed analysis of sibling relationships, maintaining database integrity and adherence to predefined student groupings.

4.3 Query 3: Instructor's Lesson Limit During Current Month

```sql
/*Query 3*/
-- View to track instructor's lessons per month
CREATE VIEW instructor_lessons_per_month AS
SELECT
    i.instructor_id,
    p.first_name,
    p.last_name,
    EXTRACT(MONTH FROM l.date) AS lesson_month,
    EXTRACT(YEAR FROM l.date) AS lesson_year,
    COUNT(*) AS number_of_lessons
FROM
    instructor i
JOIN
    lesson l ON i.instructor_id = l.instructor_id
JOIN
    person p ON i.person_id = p.person_id
GROUP BY
    i.instructor_id, p.first_name, p.last_name, lesson_month, lesson_year;

--The final query
SELECT
    instructor_id,
    first_name,
    last_name,
    number_of_lessons
FROM
    instructor_lessons_per_month
WHERE
    lesson_month = EXTRACT(MONTH FROM CURRENT_DATE)
    AND lesson_year = EXTRACT(YEAR FROM CURRENT_DATE)
    AND number_of_lessons > 3
ORDER BY
    number_of_lessons DESC;
```

instructor_id integer	first_name character varying (50)	last_name character varying (50)	number_of_lessons bigint
11	Darryl	Pate	6
16	Bianca	Schroeder	6
1	Baxter	Little	5
17	Martha	Conner	4

Figure 3: Code for Query 3 and the Resulting Table

Query 3 above aims to provide a structured view, 'instructor_lessons_per_month,' encapsulating information about instructors, including their names, instructor IDs, and the count of lessons they have conducted during a specific month. The view aggregates data from the 'instructor,' 'lesson,' and 'person' tables, linking instructors to their respective lessons and personal information. By leveraging the 'EXTRACT' function, the query breaks down lesson dates into month and year components for comprehensive grouping. The resulting view, 'instructor_lessons_per_month,' thereby serves as a concise and informative representation of instructor activity at the school.

The final query utilizes this view to extract pertinent details about instructors who have conducted more than a specified number of lessons during the current month. Filtering instructors based on the current month and year, as derived from the 'EXTRACT' function, and imposing a condition on the number of lessons, the query identifies instructors who may be at risk of excessive workload. The final output includes instructor IDs, names, and the respective count of lessons, facilitating a clear assessment of instructor workloads. This dynamic and efficient query is designed for daily execution, providing a valuable tool for identifying instructors who may need workload management attention.

4.4 Query 4: Ensemble Lessons during the upcoming week - Availability Status

```sql
/*Query 4*/
CREATE MATERIALIZED VIEW ensemble_lesson_status AS
SELECT
    TO_CHAR(l.date, 'Day') AS day_of_week,
    el.genre,
    l.date,
    CASE
        WHEN COUNT(sl.student_id) >= l.max_places THEN 'Full Booked'
        WHEN COUNT(sl.student_id) >= l.max_places - 2 THEN '1-2 Seats Left'
        ELSE 'More Seats Left'
    END AS seats_status
FROM
    ensemble_lesson el
JOIN
    lesson l ON el.lesson_id = l.lesson_id
LEFT JOIN
    student_lesson sl ON l.lesson_id = sl.lesson_id
GROUP BY
    TO_CHAR(l.date, 'Day'), el.genre, l.max_places, l.date;

-- An index on the materialized view for better performance
CREATE INDEX idx_lesson_status_date ON ensemble_lesson_status (date);
-- Refresh the materialized view
REFRESH MATERIALIZED VIEW ensemble_lesson_status;
-- The final query
SELECT
    day_of_week,
    genre,
    date,
    seats_status
FROM
    ensemble_lesson_status
WHERE
    date BETWEEN CURRENT_DATE AND CURRENT_DATE + INTERVAL '1 week'
ORDER BY
    date;
```

day_of_week text	genre character varying (10)	date date	seats_status text
Friday	orchestra	2023-12-01	1-2 Seats Left
Saturday	orchestra	2023-12-02	1-2 Seats Left
Sunday	brass	2023-12-03	1-2 Seats Left
Sunday	jaz	2023-12-03	1-2 Seats Left
Sunday	orchestra	2023-12-03	1-2 Seats Left
Monday	orchestra	2023-12-04	More Seats Left
Monday	pop	2023-12-04	Full Booked
Monday	pop	2023-12-04	More Seats Left
Monday	pop	2023-12-04	More Seats Left
Wednesday	pop	2023-12-06	More Seats Left
Wednesday	rock	2023-12-06	More Seats Left
Thursday	jaz	2023-12-07	Full Booked

Figure 4: Code for Query 4 and the Resulting Table

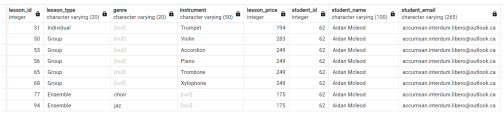
Lastly, the creation of query 4 facilitates the creation and utilization of a materialized view named 'ensemble_lesson_status' to facilitate the efficient retrieval of information regarding ensemble lessons. The use of a materialized view proves advantageous due to its ability to significantly enhance query performance and streamline the process of retrieving ensemble lesson information.Thereby, by pre-computing and storing aggregated data, the materialized view ensemble_lesson_status minimizes the computational overhead during a frequently used query.

The view is constructed by joining tables 'ensemble_lesson,' 'lesson,' and 'student_lesson,' and grouping the results by the day of the week, genre, maximum available seats in a lesson, and the lesson date. The 'CASE' statement plays a pivotal role in determining the status of available seats for each ensemble, categorizing them as 'Full Booked,' '1-2 Seats Left,' or 'More Seats Left' based on the current occupancy.

Additionally, to enhance the performance of queries involving this materialized view, an index, 'idx_lesson_status_date,' is created on the 'date' column. This index contributes to faster data retrieval when filtering or sorting based on lesson dates. The final query then leverages this materialized view to obtain a list of ensembles scheduled for the next week. The results are filtered based on the lesson dates falling within the next week, and the output includes relevant details such as the day of the week, music genre, lesson date, and the status of available seats. The query output is further organized by date to provide a comprehensive and structured overview of upcoming ensemble lessons.

This comprehensive approach to managing ensemble lesson information through a materialized view aligns with the intended use case, offering a performance-enhanced and regularly updated snapshot of lesson statuses for effective planning and scheduling.

4.5 Higher Grade Query : Historical Database



lesson_id integer	lesson_type character varying (20)	genre character varying (20)	instrument character varying (50)	lesson_price integer	student_id integer	student_name character varying (100)	student_email character varying (265)
31	Individual	[null]	Trumpet	194	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
50	Group	[null]	Violin	283	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
53	Group	[null]	Accordion	249	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
56	Group	[null]	Piano	249	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
65	Group	[null]	Trombone	249	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
68	Group	[null]	Xylophone	249	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
77	Ensemble	choir	[null]	175	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca
94	Ensemble	jaz	[null]	175	62	Aidan Mcleod	acournean.interdum.libero@outlook.ca

(a) Result Table

```sql
/* Query to select report for a student using the student name */
SELECT hl.lesson_id,hl.lesson_type,hl.genre,hl.instrument,hl.lesson_price,
hs.student_id,hs.student_name,hs.student_email
FROM hist_schema.historical_student_lesson sl
JOIN hist_schema.historical_students hs ON sl.student_id = hs.student_id
JOIN hist_schema.historical_lessons hl ON sl.lesson_id = hl.lesson_id
WHERE hs.student_name = 'Aidan Mcleod';
```

(b) Select Query

```sql
--Denormalized lesson table
CREATE TABLE hist_schema.historical_lessons (
    "id" int GENERATED ALWAYS AS IDENTITY NOT NULL PRIMARY KEY,
    lesson_id int,
    lesson_type varchar(20),
    genre varchar(20),
    instrument varchar(50),
    lesson_price int
);
INSERT INTO hist_schema.historical_lessons (
    lesson_id,
    lesson_type,
    genre,
    instrument,
    lesson_price
)
SELECT
    student_lesson.lesson_id,
    lesson_type.type AS lesson_type,
    ensemble_lesson.genre AS lesson_genre,
    CASE WHEN lesson_type.type = 'Ensemble' THEN NULL ELSE MIN(instrument_stock.instrument_name) END AS instrument_name,
    price.price AS lesson_price
FROM
    hist_schema.student_lesson student_lesson
LEFT JOIN
    hist_schema.ensemble_lesson ensemble_lesson ON student_lesson.lesson_id = ensemble_lesson.lesson_id
LEFT JOIN
    hist_schema.lesson lesson ON student_lesson.lesson_id = lesson.lesson_id
LEFT JOIN
    hist_schema.pricing_scheme price ON lesson.price_id = price.price_id
LEFT JOIN
    hist_schema.lesson_type lesson_type ON price.lessontype_id = lesson_type.lessontype_id
LEFT JOIN
    hist_schema.instrument_type instrument_type ON student_lesson.lesson_id = instrument_type.lesson_id
LEFT JOIN
```

(c) Denormalized Lesson Historical Table

```sql
--Denormalized student table
CREATE TABLE hist_schema.historical_students (
    "id" int GENERATED ALWAYS AS IDENTITY NOT NULL PRIMARY KEY,
    student_id int,
    student_name varchar(100),
    student_email varchar(265)
);
INSERT INTO hist_schema.historical_students (
    student_id,
    student_name,
    student_email
)
SELECT
    student_lesson.student_id AS student_id,
    CONCAT(person.first_name, ' ', person.last_name) AS student_name,
    e.email AS student_email
FROM
    hist_schema.student_lesson student_lesson
LEFT JOIN
    hist_schema.person person ON student_lesson.student_id = person.person_id
LEFT JOIN
    hist_schema.email e ON person.person_id = e.person_id
GROUP BY
    student_lesson.student_id, student_name, student_email
ORDER BY
    student_name;


-- Copy the existing student_lesson table
CREATE TABLE hist_schema.historical_student_lesson AS
SELECT * FROM hist_schema.student_lesson;
```

(d) Denormalized Student Info Table

Figure 5: Higher Grade Code and Result

Table c above illustrates the the creation of the de-normalised historical lesson table. The historical lessons combines data from various related tables to provide a comprehensive snapshot of historical lesson information. It includes attributes such as lesson ID, lesson type, genre, instrument, and lesson price. The SQL query utilizes multiple left joins to combine data from tables including 'student_lesson,' 'ensemble_lesson,' 'lesson,' 'pricing_scheme,' 'lesson_type,' 'instrument_type,' 'instrument,' and 'instrument_stock.

The de-normalization process involves incorporating relevant information from these tables into a single, consolidated table to simplify and optimize queries for historical lesson data. The SELECT statement aggregates data by grouping lessons based on their IDs, types, genres, and prices. Notably, the CASE statement handles special conditions, such as setting the instrument name to NULL for ensemble lessons. The resulting de-normalized table 'historical_lessons' provides a condensed and efficient representation of historical lesson details, facilitating easier and faster analysis of lesson data within the specified schema.

Similarly, historical students table is created inside the hist schema. The historical students table created inside the historical database consolidates essential student information, including unique identifiers, student IDs, names, and emails. The data is derived from related tables ('student_lesson,' 'person,' and 'email') using LEFT JOIN

operations. The resulting table facilitates more efficient and cohesive queries for historical student data, offering a streamlined representation. Each entry in the table is unique, organized by student names in alphabetical order.

The final addition to the historical database is the historical_student_lesson table, represented in figure (d). This table closely replicates the structure of the tudent_lesson table from SoundGood Music School, featuring lesson and student IDs. Its primary function is to establish a connection between students and the lessons they have enrolled in.

Lastly, the higher grade query is displayed by figure (b). The resulting table can be seen in figure (a) for this particular query. The query used to retrieve comprehensive details about the student and the lesson. This SQL query is designed to support Soundgood music school's marketing efforts by providing a detailed report on the lesson history and associated costs for individual students. Specifically, the query focuses on student "Aidan Mcleod", filtering the results based on this student's name. The de-normalized tables:'historical_student_lesson,' 'historical_students,' and 'historical_lessons'—are strategically joined using the JOIN operations, aligning with the school's requirement to maintain records of all lessons, participants, and associated prices.

5 Discussion

The views created in the queries for the Soundgood Music School database have been purposefully employed to enhance the query organisation, readability and the overall performance.

Query 1's creation of the "LessonStatistics" view is a useful way to combine essential features and reasoning so that important data about lessons taught over a given time span can be computed and displayed. This view leverages the EXTRACT function to extract year and month components from lesson dates, facilitating a meaningful temporal grouping of the data. The final query then utilizes this view to streamline the process of obtaining a concise summary of lesson statistics for each month. Leveraging the view strategically not only improves organization but also enhances readability, aligning well with the requirements for routine analysis and reporting by Soundgood music school.

In Query 2, the creation of two distinct views, namely 'SiblingCounts' and 'DistinctStudents,' showcases a deliberate effort to enhance the clarity and flexibility of the analysis. 'SiblingCounts' calculates the number of siblings for each student, while 'DistinctStudents' compiles a comprehensive list of unique student identifiers. The main query benefits from the versatile and reusable attributes of these views, facilitating a more targeted and understandable analysis of student demographics with a focus on sibling counts.

The incorporation of the instructor_lessons_per_month view in Query 3 is also a notable example of employing views for effective data consolidation. This view brings together information about instructors, such as their names, IDs, and the number of lessons

taught in a given month. By utilizing this view in the final query, a streamlined and focused analysis of instructor workloads is achieved. Thereby, serving as a valuable tool to identify teacher who may require assistance with workload management.

Furthermore, query 4 introduces a materialized view named ensemble_lesson_status to efficiently retrieve information about ensemble lessons. This materialized view, leveraging pre-computed and stored aggregated data, is designed to enhance query performance by minimizing computational overhead during frequent queries. The final query benefits from this materialized view, obtaining a regularly updated snapshot of lesson statuses for effective planning and scheduling. This strategic use aligns with the intended use case, offering a performance-enhanced and regularly updated overview of ensemble lesson statuses.

It is also evaluated that there are no unnecessarily long or complicated queries, thus, there is also no usage of a UNION clause in the discussed queries. Each query appears to be well-structured, focusing on specific tasks and utilizing appropriate clauses and views to enhance clarity and modularity. The queries are designed with efficiency and readability in mind, contributing to effective data retrieval and analysis.

It should also be noted that no changes are made to the database design to simplify these queries. The queries were developed within the existing database design, and no alterations were made to compromise the integrity or efficiency of the database structure. The queries were crafted to work seamlessly with the original database design, ensuring that the structure remains robust and suitable for various analytical tasks without any adverse impact.

Moreover, denormalizing the database for a comprehensive view of students' lessons and related expenses was required for proceeding on to the higher-grade task. This meant creating a single lesson table out of individual, group, and ensemble lessons. The goal of the denormalization process was to create a unified perspective that would make it easier to track lessons and expenses for every student starting at enrollment.

It's important to note that denormalization, as it applies to this situation, has both benefits and cons. Simplifying individual, group, and ensemble lessons into a single table improves query performance and makes complex queries simpler. It simplifies data retrieval by doing away with the necessity of intricate joins between several tables. This reorganisation increases query speed, facilitating quicker reporting and analysis.

Thereby, one notable issue is the possibility of redundant data due to the use of de-normalized tables. For example, in tables like historical_lessons, there may be repeated information, which can increase storage needs and create the risk of inconsistencies if updates are not carefully handled. This situation can lead to update anomalies, where changes made in one part of the database may not accurately reflect in other areas, potentially causing inaccuracies.

Another concern is the complexity involved in maintaining de-normalized structures, especially during updates or deletions. This complexity adds an extra layer of consideration to database management. Moreover, the design's specific optimization for certain

reporting needs might limit flexibility when accommodating new reporting requirements or adapting to changes in data structure. For instance, if there's a need for additional reporting features in the future, it may require further modifications that could affect the overall flexibility of the system.

Lastly, a comprehensive analysis was conducted on a specific query, specifically the third query, to evaluate its effectiveness. The objective was to comprehend its execution and determine the performance cost, achieved through the utilization of PostgreSQL's 'EXPLAIN' command. The resulting query plan is presented below for reference:

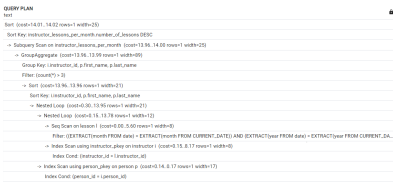


Figure 6: Query Plan for the third Query

A detailed explanation of each step in the query plan's execution process is provided. The data is first sorted according to the number of lessons in the column. Next, a GroupAggregate operation groups the data by the instructor's ID, first and last names, and filters the results by a count greater than three. The Nested Loop process appears to take the longest, combining information from the person, instructor, and lesson tables. This process entails index searches on the instructor and person tables as well as sequential scanning of the lesson table. The purpose of the query is to extract instructor information and lesson counts, with a focus on those instructors who have taught more than three lessons in the given month and year. This justifies the extensive data retrieval and joining.

It should be noted that the actual execution times may differ from the estimated costs, but considering the purpose of the query and the relationships between the involved tables, it makes sense to focus on these particular operations.