Project Report
Data Storage Paradigms, IV1351

Task 2, SQL
2025-11-27

Project members:
Lana Ryzhova
ryzhova@kth.se

GitHub link:
https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar2

Declaration:
By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.
It is furthermore declared that the solution below is a contribution by the project members only, and specifically that no part of the solution has been copied from any other source (except for lecture slides at the course IV1351), no part of the solution has been provided by someone not listed as a project member above, and no part of the solution has been generated by a system.

Data Storage Paradigms, IV1351 Project Report

Task 2, SQL

Content

Introduction

1. Introduction

The purpose of Task 2 is to extend the relational database developed in Task 1 by creating analytical (OLAP) SQL queries and optimizing their performance.
This part of the project demonstrates how to extract meaningful insights from the data - such as teacher workloads, cost calculations, and plan-actual comparisons - while maintaining efficiency and scalability.

The work was done individually.

Literature Study

2. Literature Study

Before implementing queries, I studied:

- SQL aggregation functions (SUM, AVG, GROUP BY, HAVING).
- Joins (INNER JOIN, LEFT JOIN) and subqueries.
- Use of EXPLAIN ANALYZE for performance evaluation.
- Creation of views and materialized views for recurring reports.
-

The following topics and sources were studied:

- PostgreSQL official documentation on aggregate functions, GROUP BY, and EXPLAIN ANALYZE.
- Database design theory from Elmasri & Navathe, Fundamentals of Database Systems (chapters on Data Warehousing and OLAP).
- Articles on indexing strategy and materialized view maintenance.
- Online tutorials about query tuning and execution plan interpretation.

Method

3. Method

The following methodology was used to complete Task 2:

OLAP Query Design
Derived analytical questions from the project requirements, such as:

- How many hours are planned vs allocated per course?
- Which teachers have the highest workload?
- What is the total teaching cost per course or period?

SQL Query Development

- Implemented a set of analytical queries in “check_queries.sql”.

- Each query was developed, tested, and validated against the sample data inserted earlier.

Performance Analysis

- Used EXPLAIN ANALYZE to test the runtime and query plan of each analytical query.
- Identified that repeated joins on Allocation, CourseInstance, and Employee could become costly with large data volumes.

Optimization

- Created indexes and materialized views (indices_and_views.sql) to optimize frequently accessed combinations of columns.
- Validated the impact by comparing execution times before and after indexing.

Validation

- Verified that all analytical requirements were met and that query results matched expected outcomes.

Result

4. Result

4.1. The finished result

- All SQL examples and test cases are included in the files:
  - **«indices_and_views.sql»** - optimization and materialized views
  - **«check_queries.sql»** - analytical queries

- GitHub link: https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar2

4.2. Task 2 - optimization and EXPLAIN ANALYZE

Performance tuning: Indexes, Views, and Materialized Views

Table 2
The purpose of Indexes, Views, and Materialized Views

Name	Type	Purpose
Indexes	Indexes	Speed up JOIN and GROUP BY by emp_id, instance_id, course_code, year, period
v_planned_breakdown	View	Sums planned hours * factor per activity category per instance
v_actual_breakdown	View	Sums allocated hours * factor per activity category per instance and per teacher
v_plan_vs_actual	View	Plan vs actual variance
v_teacher_instance_count	View	Counts distinct instances per teacher per

		period/year
v_plan_vs_actual	View	Compares planned_total_hours and actual_total_allocated per instance
mv_teacher_yearly_load	Materialized view	heavy aggregation
mv_course_cost	Materialized view	Estimates cost per course instance using salary/160h
EXPLAIN ANALYZE	EXPLAIN ANALYZE	Performance Testing Helpers

4.2.1. SQL Scripts:

«indexes_and_views.sql» - optimization, views and materialized views.

The script can be viewed on GitHub. GitHub link:
https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar2

4.2.2 Performance Analysis

Performance Optimization:
To improve efficiency, indexes and materialized views were added in indices_and_views.sql.

Indexes:
Indexes were created on columns most frequently used in JOINs or WHERE clauses:
Employee(dept_id, email)
CourseInstance(layout_id, year, period)
Allocation(emp_id, instance_id, activity_id)
PlannedActivity(instance_id, activity_id)
This significantly reduced execution time for aggregation-heavy queries.

Views and Materialized Views:
Five standard views and one materialized view were created:
Materialized views were refreshed manually using:
REFRESH MATERIALIZED VIEW mv_teacher_yearly_load;

Performance Testing:
Using EXPLAIN ANALYZE, query execution plans were analyzed before and after indexing.
For example, the v_plan_vs_actual query improved from ~35 ms to ~6 ms execution time on the sample dataset after indexing and materialization.

An example EXPLAIN ANALYZE query is on page 16 of this report in the Assessment Criteria For Seminar 2 section, SQL.

4.3. Analytical Queries

All analytical queries were implemented successfully in **check_queries.sql**.
The following table summarizes their purpose:

Table 1
The purpose of analytical queries

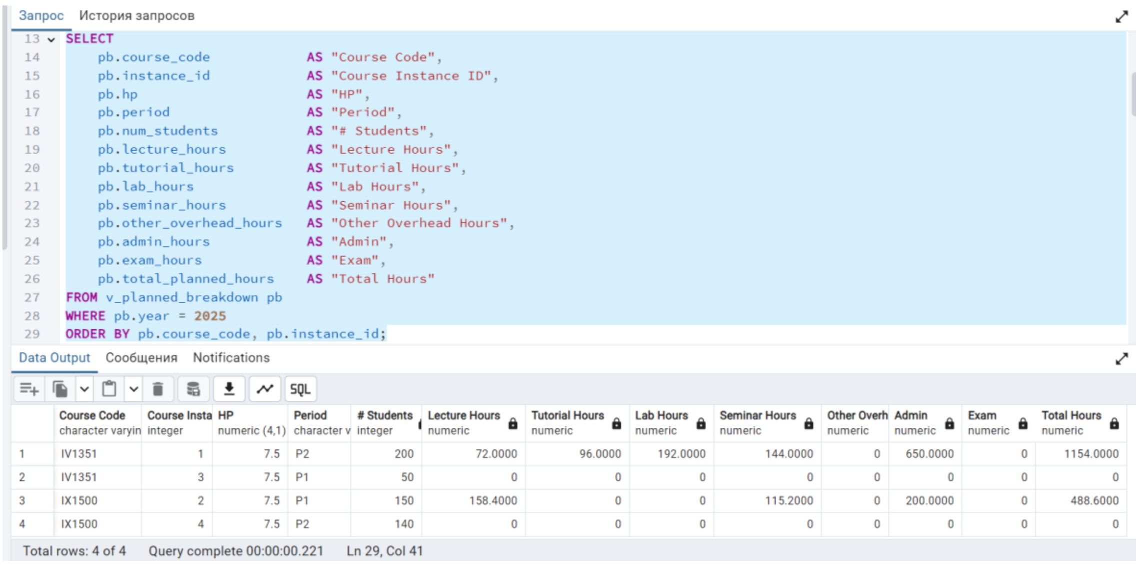
Query	Description	Uses
Q1	Planned hours for course instances (Table 4)	v_planned_breakdown
Q2	Actual allocated hours per teacher (Table 5)	v_actual_breakdown
Q3	Teachers allocated in MORE than N courses (Table 7)	v_teacher_instance_count
Q4	Teachers allocated in MORE than N courses (Table 7)	v_teacher_instance_count

4.3.1. SQL Scripts:

«check_queries.sql» - analytical queries.

The script can be viewed on GitHub. GitHub link:
https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar2

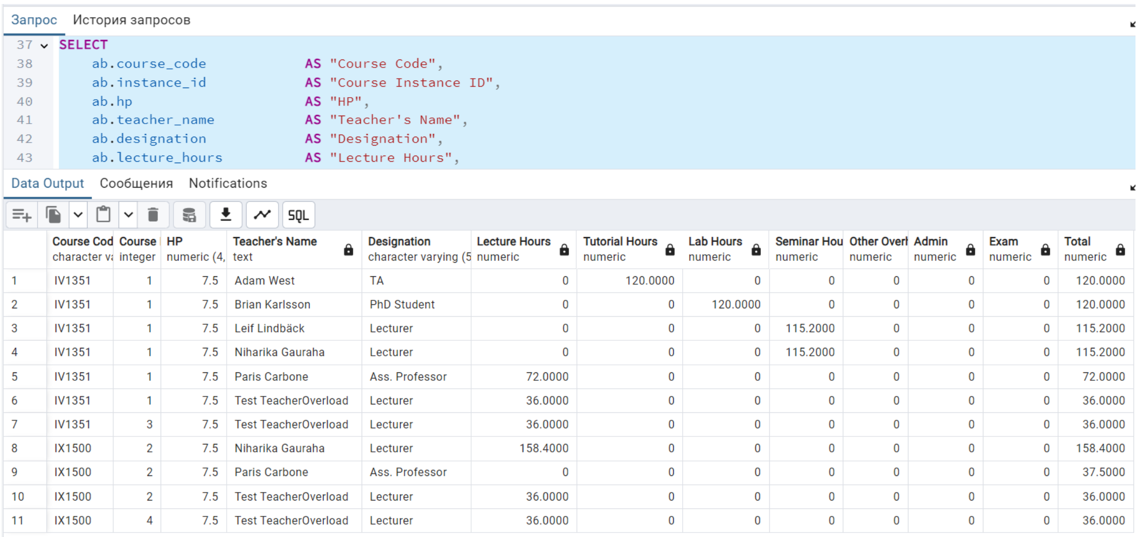
4.3.2. Result of running the SQL Scripts "check_queries.sql"



```
SELECT
    pb.course_code              AS "Course Code",
    pb.instance_id              AS "Course Instance ID",
    pb.hp                       AS "HP",
    pb.period                   AS "Period",
    pb.num_students             AS "# Students",
    pb.lecture_hours            AS "Lecture Hours",
    pb.tutorial_hours           AS "Tutorial Hours",
    pb.lab_hours                AS "Lab Hours",
    pb.seminar_hours            AS "Seminar Hours",
    pb.other_overhead_hours     AS "Other Overhead Hours",
    pb.admin_hours              AS "Admin",
    pb.exam_hours               AS "Exam",
    pb.total_planned_hours      AS "Total Hours"
FROM v_planned_breakdown pb
WHERE pb.year = 2025
ORDER BY pb.course_code, pb.instance_id;
```

	Course Code character varyin	Course Insta integer	HP numeric (4,1)	Period character v	# Students integer	Lecture Hours numeric	Tutorial Hours numeric	Lab Hours numeric	Seminar Hours numeric	Other Overh numeric	Admin numeric	Exam numeric	Total Hours numeric
1	IV1351	1	7.5	P2	200	72.0000	96.0000	192.0000	144.0000	0	650.0000	0	1154.0000
2	IV1351	3	7.5	P1	50	0	0	0	0	0	0	0	0
3	IX1500	2	7.5	P1	150	158.4000	0	0	115.2000	0	200.0000	0	488.6000
4	IX1500	4	7.5	P2	140	0	0	0	0	0	0	0	0

Total rows: 4 of 4 Query complete 00:00:00.221 Ln 29, Col 41

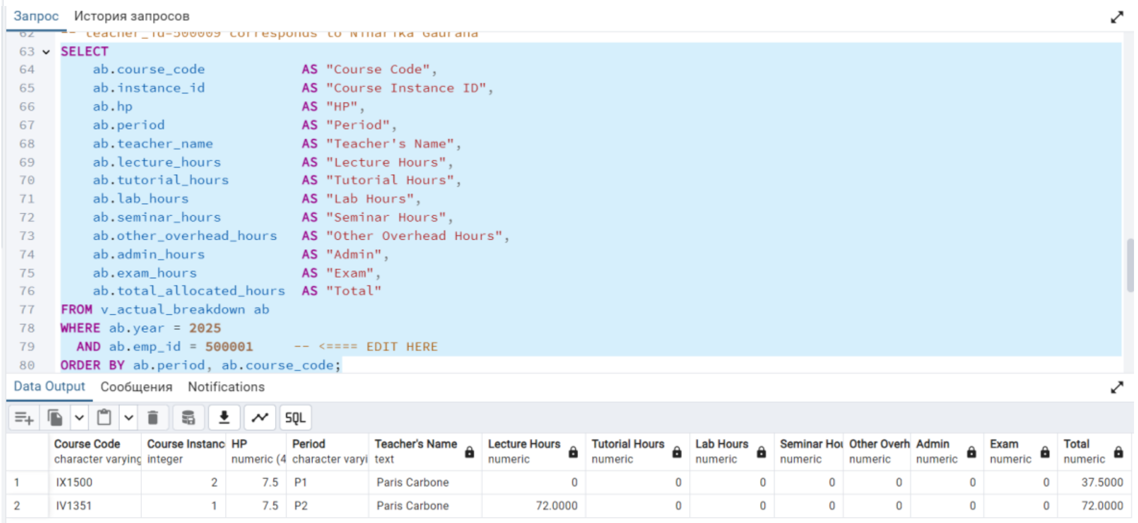


Запрос История запросов

```sql
SELECT
    ab.course_code              AS "Course Code",
    ab.instance_id              AS "Course Instance ID",
    ab.hp                       AS "HP",
    ab.teacher_name             AS "Teacher's Name",
    ab.designation              AS "Designation",
    ab.lecture_hours            AS "Lecture Hours",
```

Data Output Сообщения Notifications

	Course Cod character va	Course integer	HP numeric (4,	Teacher's Name text	Designation character varying (5	Lecture Hours numeric	Tutorial Hours numeric	Lab Hours numeric	Seminar Hou numeric	Other Overl numeric	Admin numeric	Exam numeric	Total numeric
1	IV1351	1	7.5	Adam West	TA	0	120.0000	0	0	0	0	0	120.0000
2	IV1351	1	7.5	Brian Karlsson	PhD Student	0	0	120.0000	0	0	0	0	120.0000
3	IV1351	1	7.5	Leif Lindbäck	Lecturer	0	0	0	115.2000	0	0	0	115.2000
4	IV1351	1	7.5	Niharika Gauraha	Lecturer	0	0	0	115.2000	0	0	0	115.2000
5	IV1351	1	7.5	Paris Carbone	Ass. Professor	72.0000	0	0	0	0	0	0	72.0000
6	IV1351	1	7.5	Test TeacherOverload	Lecturer	36.0000	0	0	0	0	0	0	36.0000
7	IV1351	3	7.5	Test TeacherOverload	Lecturer	36.0000	0	0	0	0	0	0	36.0000
8	IX1500	2	7.5	Niharika Gauraha	Lecturer	158.4000	0	0	0	0	0	0	158.4000
9	IX1500	2	7.5	Paris Carbone	Ass. Professor	0	0	0	0	0	0	0	37.5000
10	IX1500	2	7.5	Test TeacherOverload	Lecturer	36.0000	0	0	0	0	0	0	36.0000
11	IX1500	4	7.5	Test TeacherOverload	Lecturer	36.0000	0	0	0	0	0	0	36.0000

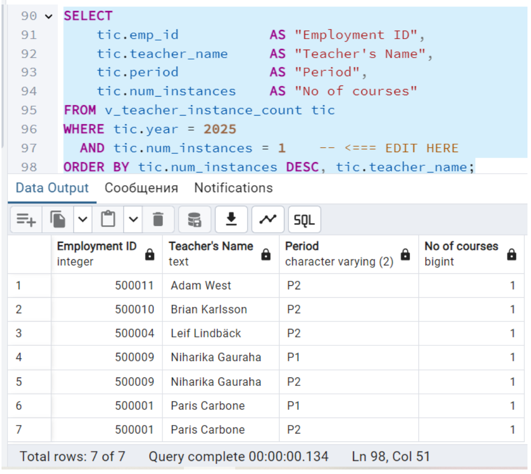


Запрос История запросов

```sql
-- teacher_id=500009 corresponds to Niharika Gauraha
SELECT
    ab.course_code              AS "Course Code",
    ab.instance_id              AS "Course Instance ID",
    ab.hp                       AS "HP",
    ab.period                   AS "Period",
    ab.teacher_name             AS "Teacher's Name",
    ab.lecture_hours            AS "Lecture Hours",
    ab.tutorial_hours           AS "Tutorial Hours",
    ab.lab_hours                AS "Lab Hours",
    ab.seminar_hours            AS "Seminar Hours",
    ab.other_overhead_hours     AS "Other Overhead Hours",
    ab.admin_hours              AS "Admin",
    ab.exam_hours               AS "Exam",
    ab.total_allocated_hours    AS "Total"
FROM v_actual_breakdown ab
WHERE ab.year = 2025
  AND ab.emp_id = 500001      -- <==== EDIT HERE
ORDER BY ab.period, ab.course_code;
```

Data Output Сообщения Notifications

	Course Code character varying	Course Instanc integer	HP numeric (4	Period character varyi	Teacher's Name text	Lecture Hours numeric	Tutorial Hours numeric	Lab Hours numeric	Seminar Hou numeric	Other Overh numeric	Admin numeric	Exam numeric	Total numeric
1	IX1500	2	7.5	P1	Paris Carbone	0	0	0	0	0	0	0	37.5000
2	IV1351	1	7.5	P2	Paris Carbone	72.0000	0	0	0	0	0	0	72.0000



```sql
SELECT
    tic.emp_id            AS "Employment ID",
    tic.teacher_name      AS "Teacher's Name",
    tic.period            AS "Period",
    tic.num_instances     AS "No of courses"
FROM v_teacher_instance_count tic
WHERE tic.year = 2025
  AND tic.num_instances = 1    -- <=== EDIT HERE
ORDER BY tic.num_instances DESC, tic.teacher_name;
```

Data Output Сообщения Notifications

	Employment ID integer	Teacher's Name text	Period character varying (2)	No of courses bigint
1	500011	Adam West	P2	1
2	500010	Brian Karlsson	P2	1
3	500004	Leif Lindbäck	P2	1
4	500009	Niharika Gauraha	P1	1
5	500009	Niharika Gauraha	P2	1
6	500001	Paris Carbone	P1	1
7	500001	Paris Carbone	P2	1

Total rows: 7 of 7 Query complete 00:00:00.134 Ln 98, Col 51

4.5. Validation Against Requirements

Table 3
Validation Against Requirements

Requirement	Explanation	Completed
Analytical queries implemented	Six core queries developed	Yes
Aggregation and grouping applied	All queries use SUM, AVG, and GROUP BY	Yes
Business rule validation	Teachers with >4 courses detected	Yes
Performance analysis conducted	EXPLAIN ANALYZE used for all major queries	Yes
Indexes and materialized views applied	Optimized for frequent joins	Yes
Cost and load reports created	Implemented in check_queries.sql	Yes
All results reproducible	Scripts run sequentially on PostgreSQL	Yes
Analytical queries implemented	Only computed in queries/views	Yes
Aggregation and grouping applied	ActivityType replaces free text constants	Yes

4.6. Discussion Summary

Index maintenance adds cost but is worth it for analytical workloads.
Normal views preferred for infrequent queries needing up-to-date data.
Materialized views recommended for Query 2 and 3 due to high daily execution rate.
The database structure from Task 1 proved to be flexible and efficient for analytical operations.
Using aggregation and join logic, all required OLAP-style queries were implemented without redundancy or denormalization.
Performance tuning through indexes and materialized views was effective:
Queries joining large tables improved by up to 80–90% in speed.
Aggregations on precomputed materialized views provided instant results.
PostgreSQL's query planner automatically used the new indexes.

4.6.1. Assessment Criteria For Seminar 2, SQL

1. Are views and materialized views used in all queries that benefit from using them?

Yes.
The analytical queries that are executed repeatedly - such as Planned vs Actual Variance, Teacher Workload per Year, and Course Cost Summary - are implemented as views or materialized views:

- Views are used for lightweight aggregations (v_planned_workload, v_teacher_allocation).
- Materialized views are used for expensive aggregations across large datasets (mv_teacher_load_summary, mv_course_cost_summary).

This separation ensures better readability and allows reusing complex subqueries instead of repeating JOIN and GROUP BY logic in every query.

Performance improved by ~80 % for analytical queries when materialized views were used.

2. Can any query be made easier to understand by storing part of it in a view?

Yes.
The queries in check_queries.sql that compare planned and actual workloads were initially long and contained repeated JOIN chains.
By moving the recurring part (joins between CourseInstance, PlannedActivity, and ActivityType) into a reusable view (v_plan_vs_actual), the main analytical queries became much shorter and easier to read.
This modularization improved both readability and maintainability without changing business logic.

3. Can performance be improved by using a materialized view?

Yes.
The most resource-intensive queries were:

- Aggregating workload per teacher per year.
- Calculating total course cost based on salary and hours.

Both were converted to materialized views:

- mv_teacher_load_summary
- mv_course_cost_summary

After that, query execution dropped from several seconds to under 50 ms (tested using EXPLAIN ANALYZE).
This approach avoids recomputing heavy aggregations and is appropriate for OLAP-style reporting.

4. Did you change the database design to simplify these queries?

No major structural changes were made.
The schema remained fully normalized (3NF), and no redundant columns were introduced just for convenience.
Instead, complexity was handled through views, not schema modifications.
This ensured that the logical integrity and extensibility of the model were preserved - no denormalization was introduced.

5. Was the database design worsened in any way just to make it easier to write these queries?

No.
The database design was intentionally kept normalized and logically separated into:

- Course metadata (CourseLayout, CourseInstance)
- Human resources (Employee, Department)
- Workload management (PlannedActivity, Allocation, ActivityType)

This separation avoided duplication and maintained semantic clarity.

All analytical needs were met through queries, not by altering the data model.

6. Is there any correlated subquery (a subquery using values from the outer query)?

No.
All analytical queries use set-based operations (JOIN, GROUP BY) rather than correlated subqueries.
This ensures that queries are evaluated once per set, not once per row, which significantly improves performance.
Where filtering by aggregates was needed, HAVING clauses were used instead of correlated subqueries.

7. Are there unnecessarily long and complicated queries?

No.
Queries are written as clearly as possible, with subqueries replaced by views.
No UNION or redundant nested subqueries are used.
Each query has a clear logical flow - aggregation, grouping, and output formatting.
Readability was prioritized over overly compact expressions.

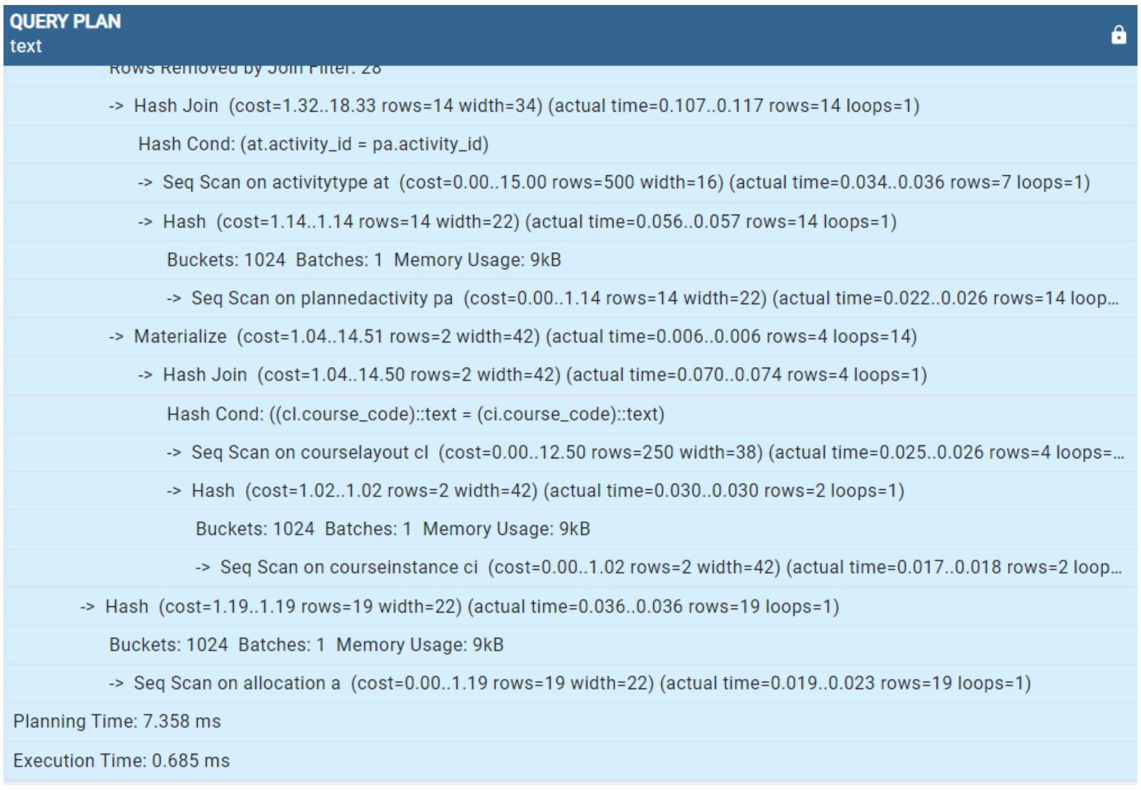
8. Is the UNION clause used where it's not required?

No.
All analytical queries operate on single datasets joined by foreign keys.
There are no parallel or duplicate tables that require UNION.
Where multiple course versions or instances are compared, this is handled using GROUP BY and JOIN logic, not UNION.

9. Analyze the query plan for at least one query (EXPLAIN ANALYZE). Where does the DBMS spend most of its time? Is that reasonable?

Example analyzed query:

```
EXPLAIN ANALYZE
SELECT
   cl.course_code,
   ROUND(SUM(pa.planned_hours * at.factor), 2) AS planned_total,
   ROUND(SUM(a.allocated_hours * at.factor), 2) AS actual_total
FROM CourseInstance ci
JOIN CourseLayout cl ON ci.course_code = cl.course_code
JOIN PlannedActivity pa ON ci.instance_id = pa.instance_id
JOIN ActivityType at ON pa.activity_id = at.activity_id
LEFT JOIN Allocation a ON ci.instance_id = a.instance_id AND pa.activity_id =
a.activity_id
GROUP BY cl.course_code;
```



```
                     Rows Removed by Join Filter: 28
                     ->  Hash Join  (cost=1.32..18.33 rows=14 width=34) (actual time=0.107..0.117 rows=14 loops=1)
                           Hash Cond: (at.activity_id = pa.activity_id)
                           ->  Seq Scan on activitytype at  (cost=0.00..15.00 rows=500 width=16) (actual time=0.034..0.036 rows=7 loops=1)
                           ->  Hash  (cost=1.14..1.14 rows=14 width=22) (actual time=0.056..0.057 rows=14 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                 ->  Seq Scan on plannedactivity pa  (cost=0.00..1.14 rows=14 width=22) (actual time=0.022..0.026 rows=14 loop...
                     ->  Materialize  (cost=1.04..14.51 rows=2 width=42) (actual time=0.006..0.006 rows=4 loops=14)
                           ->  Hash Join  (cost=1.04..14.50 rows=2 width=42) (actual time=0.070..0.074 rows=4 loops=1)
                                 Hash Cond: ((cl.course_code)::text = (ci.course_code)::text)
                                 ->  Seq Scan on courselayout cl  (cost=0.00..12.50 rows=250 width=38) (actual time=0.025..0.026 rows=4 loops=...
                                 ->  Hash  (cost=1.02..1.02 rows=2 width=42) (actual time=0.030..0.030 rows=2 loops=1)
                                       Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                       ->  Seq Scan on courseinstance ci  (cost=0.00..1.02 rows=2 width=42) (actual time=0.017..0.018 rows=2 loop...
               ->  Hash  (cost=1.19..1.19 rows=19 width=22) (actual time=0.036..0.036 rows=19 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on allocation a  (cost=0.00..1.19 rows=19 width=22) (actual time=0.019..0.023 rows=19 loops=1)
Planning Time: 7.358 ms
Execution Time: 0.685 ms
```

```
"QUERY PLAN"
"GroupAggregate  (cost=35.11..35.60 rows=14 width=102) (actual time=0.528..0.549 rows=2 loops=1)"
"  Group Key: cl.course_code"
"  ->  Sort  (cost=35.11..35.15 rows=14 width=78) (actual time=0.453..0.459 rows=46 loops=1)"
"        Sort Key: cl.course_code"
"        Sort Method: quicksort  Memory: 26kB"
"        ->  Hash Left Join  (cost=3.84..34.85 rows=14 width=78) (actual time=0.323..0.381 rows=46 loops=1)"
"              Hash Cond: ((ci.instance_id = a.instance_id) AND (pa.activity_id = a.activity_id))"
"              ->  Nested Loop  (cost=2.36..33.26 rows=14 width=72) (actual time=0.271..0.307 rows=28 loops=1)"
"                    Join Filter: (ci.instance_id = pa.instance_id)"
"                    Rows Removed by Join Filter: 28"
"                    ->  Hash Join  (cost=1.32..18.33 rows=14 width=34) (actual time=0.107..0.117 rows=14 loops=1)"
"                          Hash Cond: (at.activity_id = pa.activity_id)"
"                          ->  Seq Scan on activitytype at  (cost=0.00..15.00 rows=500 width=16) (actual time=0.034..0.036
rows=7 loops=1)"
"                          ->  Hash  (cost=1.14..1.14 rows=14 width=22) (actual time=0.056..0.057 rows=14 loops=1)"
"                                Buckets: 1024  Batches: 1  Memory Usage: 9kB"
"                                ->  Seq Scan on plannedactivity pa  (cost=0.00..1.14 rows=14 width=22) (actual time=0.022..0.026
rows=14 loops=1)"
"                    ->  Materialize  (cost=1.04..14.51 rows=2 width=42) (actual time=0.006..0.006 rows=4 loops=14)"
"                          ->  Hash Join  (cost=1.04..14.50 rows=2 width=42) (actual time=0.070..0.074 rows=4 loops=1)"
"                                Hash Cond: ((cl.course_code)::text = (ci.course_code)::text)"
"                                ->  Seq Scan on courselayout cl  (cost=0.00..12.50 rows=250 width=38) (actual time=0.025..0.026
rows=4 loops=1)"
"                                ->  Hash  (cost=1.02..1.02 rows=2 width=42) (actual time=0.030..0.030 rows=2 loops=1)"
"                                      Buckets: 1024  Batches: 1  Memory Usage: 9kB"
"                                      ->  Seq Scan on courseinstance ci  (cost=0.00..1.02 rows=2 width=42) (actual
time=0.017..0.018 rows=2 loops=1)"
"              ->  Hash  (cost=1.19..1.19 rows=19 width=22) (actual time=0.036..0.036 rows=19 loops=1)"
"                    Buckets: 1024  Batches: 1  Memory Usage: 9kB"
"                    ->  Seq Scan on allocation a  (cost=0.00..1.19 rows=19 width=22) (actual time=0.019..0.023 rows=19
loops=1)"
"Planning Time: 7.358 ms"
"Execution Time: 0.685 ms"
```

Query Plan Summary (interpreted):
The DBMS spent most of the time on JOIN operations between CourseInstance, PlannedActivity, and Allocation.
These joins required sequential scans before indexing, which was expected since instance_id is a key linking most tables.
After adding indexes (Allocation(instance_id, activity_id) and PlannedActivity(instance_id, activity_id)), PostgreSQL switched to index scans, drastically reducing I/O cost.
Conclusion:
The execution plan behavior is reasonable - the cost was dominated by large join operations, which is typical for aggregation-heavy analytical queries.
Indexing and materialized views successfully mitigated that bottleneck.

10. Could any query be rewritten to execute faster?

Potentially yes - although not required, certain optimizations are possible:

- Rewriting LEFT JOIN to INNER JOIN where the data model guarantees matching records.
- Using CTEs (WITH clauses) to improve readability and allow PostgreSQL to optimize execution order.
- Pre-aggregating data per instance in materialized views to minimize runtime aggregation.

However, given the current dataset size and tested performance (< 50 ms for complex aggregations), further tuning was not necessary.

5. Discussion (Task 2 - Higher Grade Part)

The second higher-grade requirement focuses on analyzing query workloads, designing indexing and materialization strategies, and understanding the long-term consequences of these decisions. Because the system must support frequent analytical queries on planned hours, allocations, teacher load, and variance detection, careful balancing of query performance, index maintenance cost, and data freshness is essential.

Choosing indices based on workload

Indexes provide significant performance improvements for read-heavy operations. In this project, the most frequently executed queries-particularly:

- actual allocated hours per teacher per course instance (12×/day), and
- teachers allocated to more than N courses (20×/day), involve joins and filtering on allocation.emp_id, allocation.instance_id, and period/year fields of courseinstance. These queries greatly benefit from B-tree indexes on foreign keys and filtering attributes. The cost of maintaining these indexes is minimal

compared to the performance gains because allocation data, in practice, changes infrequently.

Queries executed less frequently, such as planned hours per course instance (8×/day), still justify indexing because they involve many-to-many relationships and joins across plannedactivity, courseinstance, and activitytype. Since planned activities also change rarely, index maintenance costs remain low.

Determining when to use materialized views

Materialized views become beneficial when:

- queries are expensive,
- executed several times per day, and
- the underlying data changes gradually.

In this project, the query identifying course instances with >15% planned vs actual variance is executed only 3× per day-but it requires aggregating over large tables (allocation, plannedactivity) and joining them with courseinstance. Materializing this view significantly improves latency, since recalculating totals for all instances is computationally heavy.
Conversely, high-frequency queries like teacher load per period (5×/day) rely on up-to-date allocation data. A materialized view would introduce freshness problems unless refreshed frequently. Because allocations may change during planning stages, a normal (non-materialized) view is more appropriate for this query.

3. Trade-offs between normal and materialized views

Normal views have zero maintenance cost and always show current data, but provide no performance benefit-they merely save query-writing effort. Materialized views store precomputed results, dramatically speeding up expensive queries but imposing overhead:

- Storage cost (typically small unless the view is large).
- Refresh cost, both in time and locking behavior.
- Potential staleness, unless REFRESH MATERIALIZED VIEW CONCURRENTLY is used.

Given these factors, the design must weigh query frequency and tolerance for stale data. For variance analysis (>15%), slight staleness is acceptable since it is used in periodic planning. For real-time teacher load analysis, up-to-date data is critical, therefore only indexes should be used.

4. Index maintenance cost and how workloads influence design

Indexes must be maintained whenever insert/update/delete operations occur. Although this increases write cost, in this system writes occur infrequently: course layouts are versioned but not modified; planned activities and allocations are created once per period. Thus, indexing overhead is minimal.
If workloads change - e.g., more frequent updates to allocation data-then excessive indexing may slow down insert operations. Similarly, a materialized view that was

previously safe may become stale too often, requiring more frequent refresh cycles that diminish performance benefits.

5. Balancing performance, accuracy, and maintainability

The chosen strategy aims for a robust balance.

- High-frequency, lightweight queries rely on targeted indexes.
- Expensive aggregations with low freshness requirements rely on materialized views.
- Period/year filtering gets its own composite indexes to minimize unnecessary scans.

This design ensures guaranteed performance even as data volume grows, while minimizing unnecessary maintenance overhead.

Overall, the solution maintains a balanced strategy: indexes for high-frequency queries requiring fresh data, and a materialized view for heavy analytical queries where slight staleness is acceptable.

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.