dmbd-project

G. Alessio, A. Campagnolo

March 2024

Contents

1 Introduction

1.1 Assignment

Our assigned project involves leveraging the TPC Benchmark H to enhance and fine-tune the performance of specific queries. The focus is on optimizing execution time and boosting efficiency through the implementation of indexes and materialized views. Specifically, we are tasked with improving two distinct queries: one concerning the computation of export/import revenue, and another that necessitates a choice between assessing late delivery and calculating returned item losses.

1.2 Statistics of the data

The TPC Benchmark H database comprises eight tables, each varying in size and attribute count. The database's scale is adjustable through a scale factor (SF), which is set to 10 for this particular project. Every table is designed with a primary key and may include one or more secondary keys. This document aims to summarize the essential information, highlighting the attributes actively utilized in the project's queries. Attributes critical for query operations are marked in cyan for the query 1, olive for the query 3 and for **both are in bold**, whereas the remaining attributes serve as foreign keys, facilitating join operations.

Table	Rows	Attributes	Primary key	Total size (including PK)
Lineitem	59.986.052	16	l_orderkey, l_linenumber	9.837 GB
Orders	15.000.161	9	o_orderkey	2.3 GB
Partsupp	8.003.957	5	ps_partkey, ps_suppkey	1.5 GB
Part	2.000.012	9	p_partkey	362.88 MB
Customer	1.499.940	8	c_custkey	312.05 MB
Supplier	100.000	7	s_suppkey	20 MB
Nation	25	4	n_nationkey	24 KB
Region	5	3	r_regionkey	24 KB

Table 1: Summary of TPC Benchmark H database tables

Column	Distinct Values	Minimum	Maximum
l_orderkey	**15,000,000**	**1**	**60,000,000**
l_extendedprice	**1,351,462**	**900.91**	**104,949.50**
l_discount	**11**	**0.00**	**0.10**
l_returnflag	3	A	R
l_partkey	2,000,000	1	2,000,000
l_suppkey	100,000	1	100,000
l_linenumber	7	1	7
l_quantity	50	1.00	50.00
l_tax	9	0.00	0.08
l_shipdate	2,526	1992-01-02	1998-12-01
l_commitdate	2,466	*Not ordered*	*Not ordered*
l_receiptdate	2,555	*Not ordered*	*Not ordered*
l_comment	34,378,943	*Not ordered*	*Not ordered*
l_linestatus	2	*Not ordered*	*Not ordered*
l_shipinstruct	4	*Not ordered*	*Not ordered*
l_shipmode	7	*Not ordered*	*Not ordered*

Table 2: Statistics for Table: **lineitem**, cyan: Query1, olive: Query3

Column	Distinct Values	Minimum	Maximum
o_orderkey	**15,000,000**	**1**	**60,000,000**
o_custkey	**999,982**	**1**	**1,499,999**
o_orderdate	**2,406**	**1992-01-01**	**1998-08-02**
o_totalprice	11,944,103	838.05	558,822.56
o_shippriority	1	*Not ordered*	*Not ordered*
o_orderpriority	5	*Not ordered*	*Not ordered*
o_orderstatus	3	*Not ordered*	*Not ordered*
o_clerk	10,000	*Not ordered*	*Not ordered*
o_comment	14,097,230	*Not ordered*	*Not ordered*

Table 3: Statistics for Table: **Orders**, cyan: Query1

Column	Distinct Values	Minimum	Maximum
p_partkey	2,000,000	1	2,000,000
p_type	150	*Not ordered*	*Not ordered*
p_size	50	1	50
p_retailprice	31,681	900.91	2,098.99
p_brand	25	*Not ordered*	*Not ordered*
p_container	40	*Not ordered*	*Not ordered*
p_comment	806,046	*Not ordered*	*Not ordered*
p_name	1,999,828	*Not ordered*	*Not ordered*
p_mfgr	5	*Not ordered*	*Not ordered*

Table 4: Statistics for Table: **Part**, cyan: query1

Column	Distinct Values	Minimum	Maximum
c_custkey	**1,500,000**	**1**	**1,500,000**
c_name	1,500,000	*Not ordered*	*Not ordered*
c_nationkey	25	0	24
c_acctbal	818,834	-999.99	9999.99
c_phone	1,499,963	*Not ordered*	*Not ordered*
c_mktsegment	5	*Not ordered*	*Not ordered*
c_comment	1,496,636	*Not ordered*	*Not ordered*
c_address	1,500,000	*Not ordered*	*Not ordered*

Table 5: Statistics for Table: **Customer**, bold: query1 and query3

Column	Distinct Values	Minimum	Maximum
s_nationkey	25	0	24
s_suppkey	100,000	1	100,000
s_acctbal	95,588	-999.92	9999.93
s_comment	99,983	*Not ordered*	*Not ordered*
s_phone	100,000	*Not ordered*	*Not ordered*
s_name	100,000	*Not ordered*	*Not ordered*
s_address	100,000	*Not ordered*	*Not ordered*

Table 6: Statistics for Table: **Supplier**, cyan: query1

Column	Distinct Values	Minimum	Maximum
n_nationkey	25	0	24
n_regionkey	5	0	4
n_name	25	*Not ordered*	*Not ordered*
n_comment	25	*Not ordered*	*Not ordered*

Table 7: Statistics for Table: **Nation**, cyan: query1

Column	Distinct Values	Minimum	Maximum
r_regionkey	5	0	4
r_name	5	*Not ordered*	*Not ordered*
r_comment	5	*Not ordered*	*Not ordered*

Table 8: Statistics for Table: **Region**, cyan: query1

1.3 SQL definition of the tables

Query Schema 1 Export/import revenue value.

"Aggregation of the export/import of revenue of lineitems between two different nations (E,I) where E is the nation of the lineitem supplier and I the nations of the lineitem customer (export means that the supplier is in the nation E and import means is in the nation I). The revenue is obtained by $l_{extendedprice} * (1 - l_{discount})$ of the considered lineitems.
The aggregations should be performed with the following roll-up:
Month - Quarter - Year Type Nation - Region
The slicing is over Type and Exporting nation."

In order to assess efficiency and Performance we perform two different slices for each query.

For the query 1:

- With the slice (a) we have 14124 rows, using FRANCE with ECONOMY POLISHED TIN
- with the slice (b) we have 16140 rows, using IRAQ with ECONOMY ANODIZED STEEL

In order to select the slice I counted the most frequent ptype in the PART table and search which NATION has the most suppliers with that ptype.

For the query 3: We implement this query in order to get the median of the number of orders place by a customer:

```sql
WITH OrderCounts AS (
    SELECT
        CUSTOMER.C_NAME AS customerName,
        COUNT(ORDERS.O_ORDERKEY) AS orderCount
    FROM
        ORDERS
    JOIN
        CUSTOMER ON CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
    GROUP BY
        CUSTOMER.C_NAME
),
OrderedCounts AS (
    SELECT
        customerName,
        orderCount,
        ROW_NUMBER() OVER (ORDER BY orderCount) AS rn,
        COUNT(*) OVER () AS total
    FROM
        OrderCounts
)
SELECT
    customerName,
    orderCount
FROM
    OrderedCounts
WHERE
    rn = (total + 1) / 2 OR (total % 2 = 0 AND rn = total / 2 + 1);
```

Listing 1: Query to determine the median of the number of orders place by a customer

And we end up selecting:

- With the slice (a) we have 19 rows, using Customer#000029326
- with the slice (b) we have 14 rows, using Customer#000371519

The query A has been implemented in SQL in the following way:

```sql
SELECT
    inat.n_name as import_nation,
    ireg.r_name as import_regin,
    enat.n_name as export_nation,
    ereg.r_name as export_region,
    SUM(L.l_extendedprice * (1 - L.l_discount)) AS revenue,
    DATE_PART('month', O.o_orderdate) AS order_month,
    DATE_PART('quarter', O.o_orderdate) AS order_quarter,
    DATE_PART('year', O.o_orderdate) AS order_year,
    P.p_type AS ptype
FROM
    LINEITEM AS L
JOIN
    ORDERS AS O ON L.l_orderkey = O.o_orderkey
JOIN
    PART AS P ON P.p_partkey = L.l_partkey
JOIN
    SUPPLIER AS S ON S.s_suppkey = L.l_suppkey
```

```
JOIN
    CUSTOMER AS C ON C.c_custkey = O.o_custkey
JOIN
    NATION AS enat ON enat.n_nationkey = S.s_nationkey
JOIN
    NATION AS inat ON inat.n_nationkey = C.c_nationkey
JOIN
    REGION AS ereg ON ereg.r_regionkey = enat.n_regionkey
JOIN
    REGION AS ireg ON ireg.r_regionkey = inat.n_regionkey
WHERE
    enat.n_name = 'FRANCE'
    AND inat.n_name != enat.n_name
    AND P.p_type = 'ECONOMY POLISHED TIN'
GROUP BY
    ROLLUP(P.p_type),
    ROLLUP(ereg.r_name, enat.n_name),
    ROLLUP(ireg.r_name, inat.n_name),
    ROLLUP(DATE_PART('year', O.o_orderdate), DATE_PART('quarter', O
    .o_orderdate), DATE_PART('month', O.o_orderdate));
```

Listing 2: Query 1

Query Schema 3 Returned item loss

"The query gives the revenue loss for customers who might be having problems with the parts that are shipped to them. Revenue lost is defined as $sum(l_{extendedprice} * (1 - l_{discount}))$ for all qualifying lineitems.
The aggregations should be performed with the following roll-up:
Month - Quarter - Year , Customer
The query can be issued with the following slicing (combined):
Name of a customer, A specific quarter"

The query has been implemented in SQL in the following way:

```
SELECT
    COALESCE(CAST(DATE_PART('year', O_ORDERDATE) AS TEXT), 'Total')
     AS orderYear,
    DATE_PART('quarter', O_ORDERDATE) AS orderQuarter,
    DATE_PART('month', O_ORDERDATE) AS orderMonth,
    C_NAME AS customer,
    SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS totalRevenueLoss
FROM
    LINEITEM
JOIN
    ORDERS ON LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
JOIN
    CUSTOMER ON CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
WHERE
    L_RETURNFLAG = 'R'
    AND DATE_PART('year', O_ORDERDATE) = 1994
    AND DATE_PART('quarter', O_ORDERDATE) = 4
    AND C_NAME = 'Customer#000029326'
GROUP BY
    customer,
```

```sql
    ROLLUP(orderYear, orderQuarter, orderMonth)
ORDER BY
    customer,
    COALESCE(CAST(DATE_PART('year', O_ORDERDATE) AS TEXT), 'Total'),
    orderQuarter,
    orderMonth;
```

Listing 3: Query 3

1.4 Measurements

To assess the computational efficiency and resource utilization of query executions, we focused on two primary performance metrics, using the PostgreSQL `time` command in the following manner:

```
time psql -U username -d database_name -a -f query.sql
```

Processor Time (CPU Time): This metric measures the total duration the CPU spends processing the query, encompassing both the time directly spent executing the query (*user time*) and the time spent on system calls made by the query (*system time*). It is a critical indicator of the computational demands of the query and its efficiency in using CPU resources, excluding external delays.

Elapsed Time (Real Time): This metric records the total time elapsed from the beginning to the end of the command's execution. It includes all forms of delays, such as disk I/O, network wait times, and other external factors, providing a comprehensive view of the actual time taken for the query execution.

To ensure a comprehensive assessment and statistical significance, the `time` command was executed ten times for each query. This methodology was adopted to accumulate reliable data while efficiently managing the experiment's duration. Subsequently, we calculated the mean and standard deviation for both the real and CPU times, offering a detailed insight into the performance characteristics of query execution.

1.5 Hardware implementation

For our experiments, hardware resources were provisioned from the DigitalOcean Platform, configured with the following specifications: The virtual CPU (vCPU) utilized was a **DO-Premium-Intel** model, operating at a frequency of **4988.27 MHz**, comprising 1 core and 1 thread. The system was equipped with **2 GB of RAM**, ensuring adequate memory for the tasks at hand. Data storage was facilitated by a **70 GB NVMe SSD**, chosen for its high-speed data transfer capabilities. Graphics processing was managed by a Virtio GPU, which is virtualized to efficiently share resources in a cloud environment. The operating system installed on this setup was Ubuntu **22.04.4 LTS x86_64**, offering a stable and reliable environment for conducting our research.

2 Database not optimized

The first thing we have tested is the execution time of the queries on the database without any optimization. In order to do this we run the previous queries without any index or materialized view. The size of the database in this case is **14GB**. The results of the measurements are reported in the following tables and box-plots:

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	29.526	29.655	1.206	0.215	0.023
b	28.873	29.253	2.075	0.190	0.017

Table 9: Query 1, baseline: no optimization

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	9.547	9.518	0.415	0.059	0.012
b	9.907	10.126	0.799	0.057	0.012

Table 10: Query 3, baseline: no optimization

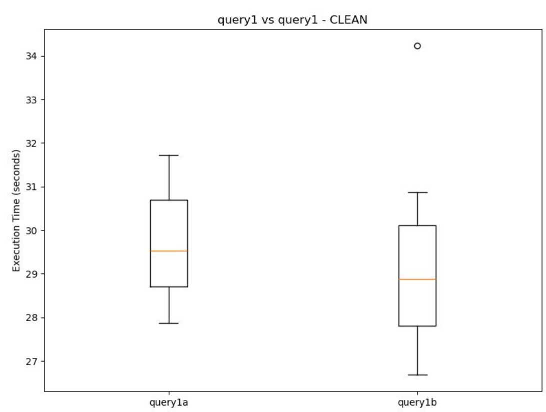

Figure 1: Box plots of the query1 slice A and slice B over the database not optimized

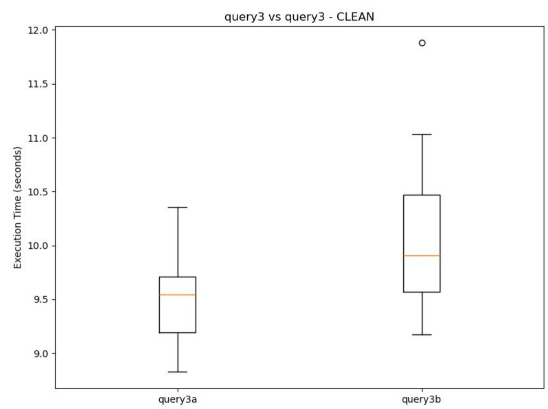

Figure 2: Box plots of the query3 slice A and slice B over the database not optimized

3 Design of indexes

We have created an index on each attribute used for queries, then we have run the queries and analyzed the query execution plan in order to understand which indexes the optimizer decided to use, with the following results:

Attribute	Table	Used?	Weight if used
`l_partkey`	LINEITEM	YES (Q1)	430 MB
`s_nationkey`	SUPPLIER	YES (Q1)	704 KB
`o_custkey`	ORDERS	YES (Q3)	120 MB
`l_orderkey`	LINEITEM	YES (Q1, Q3)	742 MB
`c_name`	CUSTOMER	YES (Q1, Q3)	58 MB
`p_type`	PART	YES (Q1)	14 MB
`l_suppkey`	LINEITEM	NO	-
`c_nationkey`	CUSTOMER	NO	-
`n_regionkey`	NATION	NO	-

Table 11: Index usage and weights for attributes in queries Q1 and Q3

Subsequently, indexes not utilized were eliminated, and execution times were reassessed. The cumulative space occupied by the employed indexes totals 2.068 GB (**space cost of indexing**). Detailed findings are presented in the subsequent tables (**query cost of queries with indexes**):

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	28.181	28.640	1.715	0.208	0.020
b	28.963	29.270	1.464	0.222	0.030

Table 12: Query 1, Indexing optimization

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	0.059	0.067	0.022	0.050	0.004
b	0.060	0.061	0.006	0.051	0.009

Table 13: Query 3, Indexing optimization

Indexing has increased memory usage by 2 GB, bringing the total space required to 16 GB. This remains within the allowable limit, which is 1.5 times the original database size. Below we present the relatives box-plots:

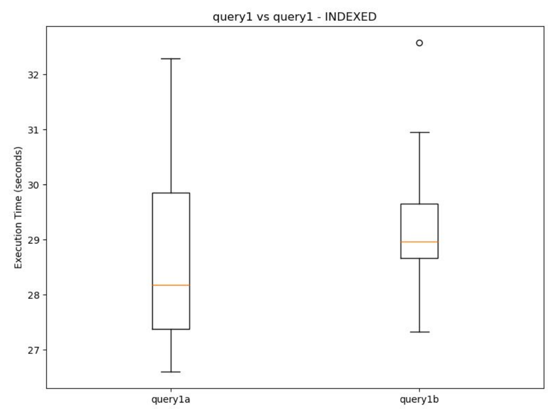

Figure 3: Box plots of the query1 slice A and slice B over the database with indexed

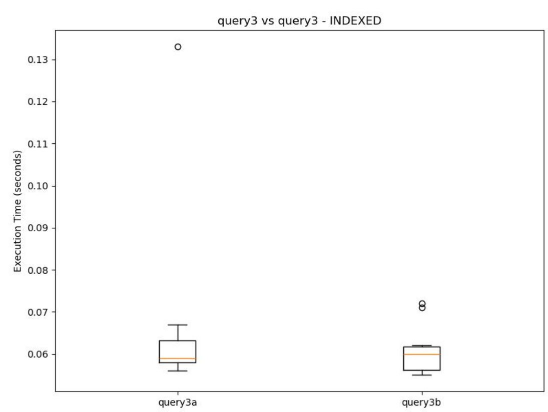

Figure 4: Box plots of the query3 slice A and slice B over the database with indexed

4 Design of materialization (without indexing)

As second optimization step, we tried to create a materialized view. In particular, we have tested two different views, one optimize it for the first query, the other optimize it for the second query:

4.1 Materialized View for Query1

The materialized view mv_q1 is tailored to optimize the execution of our first query. By precomputing the join operations and filtering the data specifically for the conditions of Query 1, this view is designed to expedite data retrieval processes that would otherwise require time-consuming computation at query time.

```
CREATE MATERIALIZED VIEW mv_q1 AS
SELECT
    inat.n_nationkey as import_nation,
    ireg.r_regionkey as import_region,
    enat.n_nationkey as export_nation,
    ereg.r_regionkey as export_region,
    O.o_orderdate AS orderdate,
    P.p_type AS ptype,
    L.l_extendedprice * (1 - L.l_discount) AS revenue
FROM
    LINEITEM AS L
JOIN
    ORDERS AS O ON L.l_orderkey = O.o_orderkey
JOIN
    PART AS P ON P.p_partkey = L.l_partkey
JOIN
    SUPPLIER AS S ON S.s_suppkey = L.l_suppkey
JOIN
    CUSTOMER AS C ON C.c_custkey = O.o_custkey
JOIN
    NATION AS enat ON enat.n_nationkey = S.s_nationkey
JOIN
    NATION AS inat ON inat.n_nationkey = C.c_nationkey
JOIN
    REGION AS ereg ON ereg.r_regionkey = enat.n_regionkey
JOIN
    REGION AS ireg ON ireg.r_regionkey = inat.n_regionkey
WHERE
    inat.n_nationkey != enat.n_nationkey;
```

Listing 4: MV for Query1

The materialized view mv_q1 is optimal for the given query because it precomputes and stores the results of complex joins, filters, and calculations, leading to faster and more efficient query execution.

4.1.1 Query1 adjusted

The adjusted Query 1 takes advantage of the pre-aggregated and pre-joined data within the mv_query1. By querying directly from the materialized view, we bypass the overhead associated with on-the-fly calculations, thereby expecting a significant reduction in query execution time.

```
SELECT
    year,
    quarter,
    month,
    part_type,
    exporting_nation_name,
    exporting_region_name,
    SUM(total_revenue) AS total_revenue
FROM
```

```sql
    mv_query1
GROUP BY ROLLUP(
    year,
    quarter,
    month,
    part_type,
    exporting_nation_name,
    exporting_region_name)
ORDER BY
    year,
    quarter,
    month,
    part_type,
    exporting_nation_name,
    exporting_region_name;
```

Listing 5: Query1 adjusted

4.1.2 Measurements

The measurement table captures the performance metrics for Query 1 utilizing the materialized view

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	22.128	20.403	4.534	0.216	0.017
b	14.918	15.232	1.477	0.225	0.021

Table 14: Query1 adjusted, MV optimization

Below, we can see the results represented through box plots:

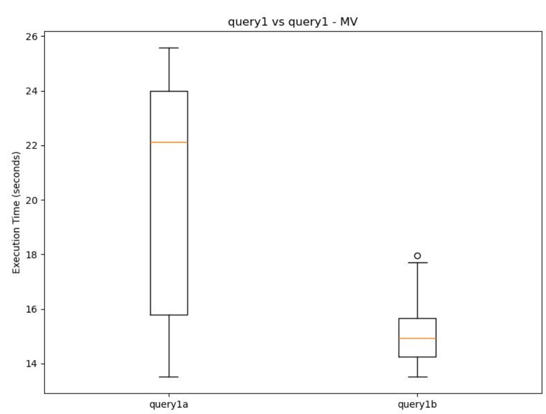

Figure 5: Box plots of the query1 slice A and slice B over the materialized view mv_q1 without index

4.2 Materialized View for Query3

Similar to the first view, mv_query3 is optimized for the third query. It focuses on pre-aggregating the specific subset of data relevant to the conditions of Query 3, potentially offering substantial performance gains due to the reduced complexity at query time.

```
CREATE MATERIALIZED VIEW mv_q3 AS
SELECT
    o_orderdate,
    l_extendedprice * (1 - l_discount) AS revenue,
    c_name
FROM
    orders
JOIN
    lineitem ON l_orderkey = o_orderkey
JOIN
    customer ON c_custkey = o_custkey
WHERE
    l_returnflag = 'R';
```

Listing 6: MV for Query3

4.2.1 Query3 adjusted

Leveraging mv_query3, the adjusted Query 3 is expected to perform with increased speed. By operating on a dataset that has already been shaped to the query's requirements, the database can more rapidly deliver the desired results.

```
SELECT
    COALESCE(CAST(DATE_PART('year', o_orderdate) AS TEXT), 'Total')
     AS orderYear,
    DATE_PART('quarter', o_orderdate) AS orderQuarter,
    DATE_PART('month', o_orderdate) AS orderMonth,
    c_name AS customer,
    SUM(revenue) AS totalRevenueLoss
FROM
    mv_q3
WHERE
    DATE_PART('year', o_orderdate) = 1994
    AND DATE_PART('quarter', o_orderdate) = 4
    AND c_name = 'Customer#000029326'
GROUP BY
    customer,
    ROLLUP(orderYear, orderQuarter, orderMonth)
ORDER BY
    customer,
    COALESCE(CAST(DATE_PART('year', o_orderdate) AS TEXT), 'Total')
    ,
    orderQuarter,
    orderMonth;
```

Listing 7: Query3 adjusted

4.2.2 Measurements

The measurement table captures the performance metrics for Query 3 utilizing the materialized view

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	1.924	1.984	0.298	0.059	0.014
b	1.832	1.888	0.145	0.061	0.01

Table 15: Query 3, MV optimization

Below, we can see the results represented through box plots:

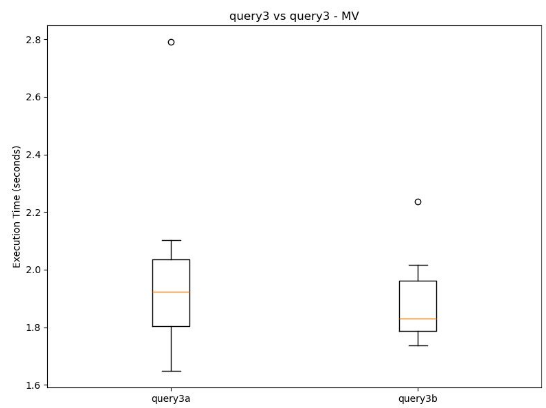

Figure 6: Box plots of the query3 slice A and slice B over the materialized view mv_q3 without index

5 Design of materialization and indexing

To enhance the efficiency of our queries, we implemented a strategy of index creation on the materialized views. This process entailed constructing indexes on every attribute within these views, executing the queries, and examining the execution plans to discern which indexes were actively leveraged. Upon identifying the utilized indexes, we streamlined the system by removing any superfluous indexes that the query optimizer did not employ. This selective refinement aimed to balance the maintenance overhead of indexing with the performance benefits they offer during query execution.

5.0.1 Indexing selection

We identified six potentially useful attributes and then kept only three of them that were actually used when executing the queries:

Attribute	Table	Used	Weight
`export_nation`	mv_q1	Q1	381 MB
`ptype`	mv_q1	Q1	391 MB
`c_name`	mv_q3	Q3	136 MB

Table 16: Index usage and weights for MV for Query1

5.0.2 Measurements

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	21.629	21.951	4.527	0.205	0.026
b	14.887	15.078	0.967	0.222	0.029

Table 17: Query 1, MV and indexing optimization

Slice	Median	Mean(s)	Standard Deviation	CPU Mean	CPU sd
a	2.086	2.383	0.677	0.062	0.015
b	2.069	2.100	0.231	0.058	0.013

Table 18: Query 3, MV and indexing optimization

Below, the box plots:

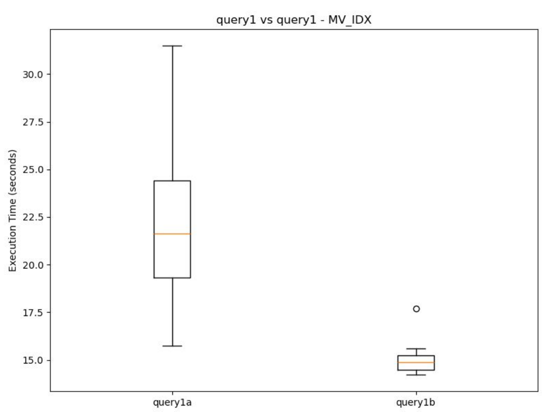

Figure 7: Box plots of the query1 slice A and slice B over the materialized view mv_q1 with index

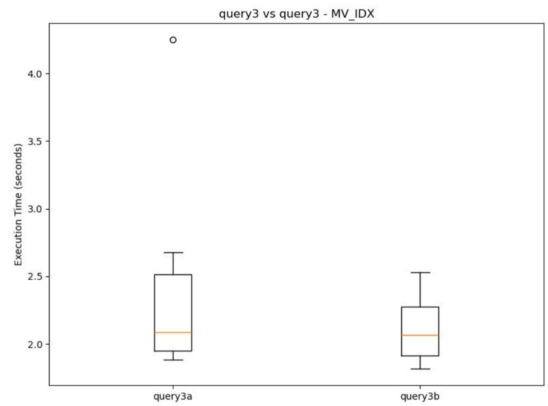

Figure 8: Box plots of the query3 slice A and slice B over the materialized view mv_q3 with index

6 Recap of the final optimization strategy

The subsequent tables encapsulate the performance outcomes for each query under varied optimization trials. Given the negligible influence of slicing values on execution times, as discerned from preceding results, we have simplified our analysis by consolidating all slicing values to compute the overall mean and median. Additionally, we provide the database size for each scenario to assess the optimization strategies' effects on storage requirements.

This streamlined approach affords a clear comparative view, elucidating the efficacy of each optimization in enhancing query performance while also considering the corresponding storage footprint.

Our initial database takes 14 GB so, with the use of all optimization technique, we needed to stay within 21 GB to respect the space constraint set by the exercise. The table below presents a recap of all the above mentioned optimization solutions.

Query	Optimization	Mean Real (s)	Std Dev Real (s)	Mean CPU (s)	Std Dev CPU (s)	DB_size(GB)	%extra
QUERY1A	CLEAN	29.655	1.206	0.215	0.023	14.000	1.000
	INDEXED	28.640	1.715	0.208	0.021	16.000	1.143
	MV	**20.403**	4.534	0.216	0.017	18.461	1.318
	MV_IDX	21.951	4.527	0.205	0.026	19.215	1.372
QUERY1B	CLEAN	29.253	2.075	0.191	0.018	14.000	1.000
	INDEXED	29.270	1.464	0.222	0.030	16.000	1.143
	MV	15.232	1.477	0.225	0.022	18.461	1.318
	MV_IDX	**15.078**	0.967	0.222	0.029	19.215	1.372
QUERY3A	CLEAN	9.518	0.415	0.059	0.012	14.000	1.000
	INDEXED	**0.067**	0.022	0.050	0.004	16.000	1.143
	MV	1.984	0.298	0.059	0.014	14.831	1.059
	MV_IDX	2.383	0.677	0.062	0.015	14.964	1.069
QUERY3B	CLEAN	10.126	0.799	0.057	0.012	14.000	1.000
	INDEXED	**0.061**	0.006	0.051	0.009	16.000	1.143
	MV	1.888	0.145	0.061	0.010	14.831	1.059
	MV_IDX	2.100	0.231	0.058	0.013	14.964	1.069

Figure 9: This is the recap table highlighting the best performance for each slice performed

The histograms below summarize the execution times (real and CPU) obtained for each optimization.

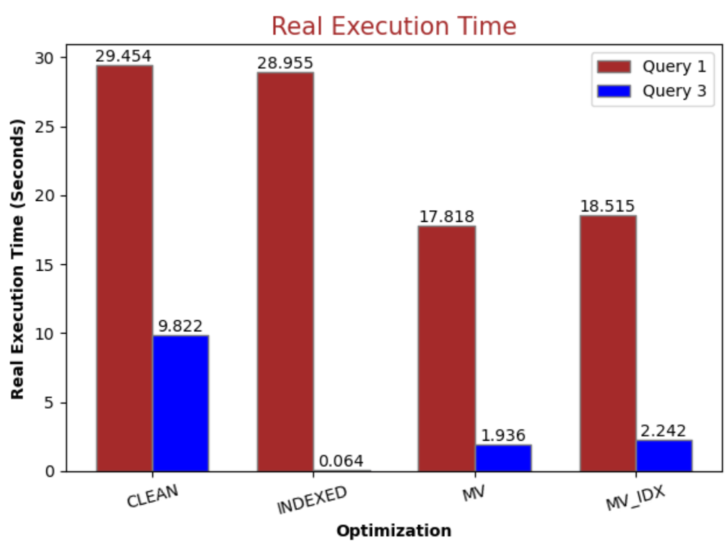

Figure 10: Recap of all optimization techniques (Real execution time)

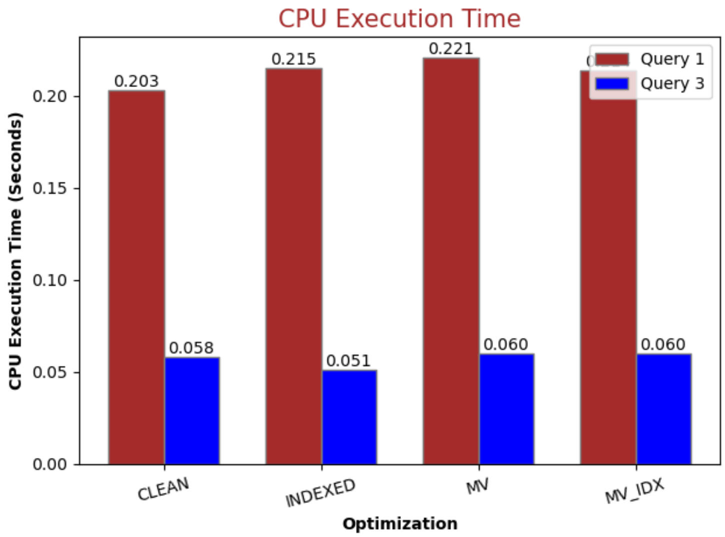

Figure 11: Recap of all optimization techniques (CPU execution time)

The most effective strategy, considering both space cost and execution time, was to use appropriate indexes on the original tables for query3 and an optimized materialized view for query1. Database optimization is about finding the right balance between efficient use of system resources and enhanced performance. Our research investigated various strategies, including adding complexity through materialized views and indexes to the database schema. It's important to always consider the overhead introduced by both indexes and materialized views. For materialized views, our primary goal was to minimize the space they occupied while maximizing the operations they precomputed for the queries. With indexes, we observed that their positive impact was significant when defined on attributes with a high selectivity rate. Ultimately, a thorough understanding of the data is essential for designing the most suitable solutions.

Furthermore, we performed two slicing operations for each query to thoroughly evaluate the effectiveness of our optimization strategies under different conditions. This helped us understand how different segments of data impact performance and identify the optimal slicing strategy.

Ultimately, a thorough understanding of the data is essential for designing the most suitable solutions. Slicing not only helps in optimizing performance but also in gaining deeper insights into data behavior under different conditions, leading to more informed and effective optimization decisions.