Data Management For Big Data

Simone Cappiello e Silvia Imeneo

March 2024

1 Introduction

1.1 Problem Statement

The aim of this case study is to implement and optimize two query schemata on the TPC Benchmark^{TM}H (TPC-H), by the use of indexes and materialized views, which have been applied both individually and together.
The query schemata that we chose are the one on export/import revenue value, and the one about returned item loss. As per the exercise's request, the implementation has been carried on in PostgreSQL.

1.2 Performance Metrics

We assessed the performance of queries' execution using the `time` command of PostgreSQL.

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

The `time` command provides two main metrics:

- **Real Time:** Represents the actual elapsed time from the start of the command to its completion. It includes all time, such as disk I/O, waiting for network responses, and other external factors.
- **CPU Time:** Indicates the total time spent by the CPU executing the query. It consists of both the time spent by the CPU executing the query (user time) and the time spent by the CPU in executing operating system calls on behalf of the query (system time). It offers insights into the computational resources consumed by the query execution process itself, excluding external factors. It helps in assessing the efficiency of the query execution algorithm and CPU resource utilization.

We conducted all experiments by executing the `time` command 30 times and subsequently calculated the mean and standard deviations for both real and CPU time.

1.3 Hardware

All the measures present in this report were collected on a laptop with these specifics:

- **Processors:** 20 × 12th Gen Intel® Core™ i7-12700H.
- **RAM:** 2 × 16 GB DDR4 3200 MT/s.
- **Graphics:** Mesa Intel® Graphics.
- **SSD:** 2 × 256 GB M.2 2280 for NVME (PCI-Express 4.0 x4), Intel Optane Technology.
- **Operating System:** Tuxedo OS 2, based on Ubuntu 22.04 LTS.

2 SQL definition of the tables

Data has been generated using a scale factor of 10. The eight tables have been created following the structure indicated by TPC-H and their SQL definition is reported below:

```sql
CREATE TABLE NATION  (
    N_NATIONKEY      INTEGER NOT NULL,
    N_NAME           CHAR(25) NOT NULL,
    N_REGIONKEY      INTEGER NOT NULL,
    N_COMMENT        VARCHAR(152));
```

```sql
CREATE TABLE REGION  (
    R_REGIONKEY      INTEGER NOT NULL,
    R_NAME           CHAR(25) NOT NULL,
    R_COMMENT        VARCHAR(152));
```

```sql
CREATE TABLE PART  (
    P_PARTKEY        INTEGER NOT NULL,
    P_NAME           VARCHAR(55) NOT NULL,
    P_MFGR           CHAR(25) NOT NULL,
    P_BRAND          CHAR(10) NOT NULL,
    P_TYPE           VARCHAR(25) NOT NULL,
    P_SIZE           INTEGER NOT NULL,
    P_CONTAINER      CHAR(10) NOT NULL,
    P_RETAILPRICE    DECIMAL(15,2) NOT NULL,
    P_COMMENT        VARCHAR(23) NOT NULL );
```

```sql
CREATE TABLE SUPPLIER (
    S_SUPPKEY        INTEGER NOT NULL,
    S_NAME           CHAR(25) NOT NULL,
    S_ADDRESS        VARCHAR(40) NOT NULL,
    S_NATIONKEY      INTEGER NOT NULL,
    S_PHONE          CHAR(15) NOT NULL,
    S_ACCTBAL        DECIMAL(15,2) NOT NULL,
    S_COMMENT        VARCHAR(101) NOT NULL);
```

```sql
CREATE TABLE PARTSUPP (
    PS_PARTKEY      INTEGER NOT NULL,
    PS_SUPPKEY      INTEGER NOT NULL,
    PS_AVAILQTY     INTEGER NOT NULL,
    PS_SUPPLYCOST   DECIMAL(15,2)  NOT NULL,
    PS_COMMENT      VARCHAR(199) NOT NULL );
```

```sql
CREATE TABLE CUSTOMER (
    C_CUSTKEY       INTEGER NOT NULL,
    C_NAME          VARCHAR(25) NOT NULL,
    C_ADDRESS       VARCHAR(40) NOT NULL,
    C_NATIONKEY     INTEGER NOT NULL,
    C_PHONE         CHAR(15) NOT NULL,
    C_ACCTBAL       DECIMAL(15,2)   NOT NULL,
    C_MKTSEGMENT    CHAR(10) NOT NULL,
    C_COMMENT       VARCHAR(117) NOT NULL);
```

```sql
CREATE TABLE ORDERS  (
    O_ORDERKEY          INTEGER NOT NULL,
    O_CUSTKEY           INTEGER NOT NULL,
    O_ORDERSTATUS       CHAR(1) NOT NULL,
    O_TOTALPRICE        DECIMAL(15,2) NOT NULL,
    O_ORDERDATE         DATE NOT NULL,
    O_ORDERPRIORITY     CHAR(15) NOT NULL,
    O_CLERK             CHAR(15) NOT NULL,
    O_SHIPPRIORITY      INTEGER NOT NULL,
    O_COMMENT           VARCHAR(79) NOT NULL);
```

```sql
CREATE TABLE LINEITEM (
    L_ORDERKEY          INTEGER NOT NULL,
    L_PARTKEY           INTEGER NOT NULL,
    L_SUPPKEY           INTEGER NOT NULL,
    L_LINENUMBER        INTEGER NOT NULL,
    L_QUANTITY          DECIMAL(15,2) NOT NULL,
    L_EXTENDEDPRICE     DECIMAL(15,2) NOT NULL,
    L_DISCOUNT          DECIMAL(15,2) NOT NULL,
    L_TAX               DECIMAL(15,2) NOT NULL,
    L_RETURNFLAG        CHAR(1) NOT NULL,
    L_LINESTATUS        CHAR(1) NOT NULL,
    L_SHIPDATE          DATE NOT NULL,
    L_COMMITDATE        DATE NOT NULL,
    L_RECEIPTDATE       DATE NOT NULL,
    L_SHIPINSTRUCT      CHAR(25) NOT NULL,
    L_SHIPMODE          CHAR(10) NOT NULL,
    L_COMMENT           VARCHAR(44) NOT NULL);
```

Each table has been populated using the personalized version of the following command:

```
\copy supplier FROM '...\supplier.csv' WITH (FORMAT csv, DELIMITER '|')
```

Primary and foreign keys have eventually been added with:

```sql
ALTER TABLE REGION
    ADD PRIMARY KEY (R_REGIONKEY);
```

```sql
ALTER TABLE PART
    ADD PRIMARY KEY (P_PARTKEY);

ALTER TABLE NATION
    ADD PRIMARY KEY (N_NATIONKEY);

ALTER TABLE SUPPLIER
    ADD PRIMARY KEY (S_SUPPKEY);
    ADD CONSTRAINT SUPPLIER_FK1
    FOREIGN KEY (S_NATIONKEY) references NATION;

ALTER TABLE CUSTOMER
    ADD PRIMARY KEY (C_CUSTKEY);
    ADD CONSTRAINT CUSTOMER_FK1
    FOREIGN KEY (C_NATIONKEY) references NATION;

ALTER TABLE ORDERS
    ADD PRIMARY KEY (O_ORDERKEY);
    ADD CONSTRAINT ORDERS_FK1
    FOREIGN KEY (O_CUSTKEY) references CUSTOMER;

ALTER TABLE PARTSUPP
    ADD PRIMARY KEY (PS_PARTKEY,PS_SUPPKEY);
    ADD CONSTRAINT PARTSUPP_FK1
    FOREIGN KEY (PS_SUPPKEY) references SUPPLIER;
    ADD CONSTRAINT PARTSUPP_FK2
    FOREIGN KEY (PS_PARTKEY) references PART;

ALTER TABLE LINEITEM
    ADD PRIMARY KEY (L_ORDERKEY,L_LINENUMBER);
    ADD CONSTRAINT LINEITEM_FK1
    FOREIGN KEY (L_ORDERKEY) references ORDERS;
    ADD CONSTRAINT LINEITEM_FK2
    FOREIGN KEY (L_PARTKEY,L_SUPPKEY) references PARTSUPP;
```

3 Statistics of the data

Below the information requested about the whole dataset and about each individual table. Regarding the details of the tables, we considered only the attributes used for querying.

ALL TABLES

	Table size (including primary keys)	Number of attributes	Number of rows
LINEITEM	11.14 GB	16	59,986,052
ORDERS	2.3 GB	9	15,000,000
PARTSUPP	1.5 GB	5	8,000,000
PART	362.88 MB	9	2,000,000
CUSTOMER	312.05 MB	8	15,000,000
SUPPLIER	19.47 MB	7	100,000
REGION	24 KB	3	5
NATION	32 KB	4	25
TOTAL	**15.6 GB**		

LINEITEM

	# Distinct values	Min Value	Max Value
L_ORDERKEY	15,000,000	1	60,000,000
L_PARTKEY	2,000,000	1	2,000,000
L_SUPPKEY	100,000	1	100,000
L_EXTENDEDPRICE	1,351,462	900.91	104,949.50
L_DISCOUNT	11	0.00	0.10
L_RETURNFLAG	3	A	R

ORDERS

	# Distinct values	Min Value	Max Value
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

PART

	# Distinct values	Min Value	Max Value
P_PARTKEY	2,000,000	0	2,000,000
P_TYPE	150	ECONOMY ANODIZED BRASS	STANDARD POLISHED TIN

CUSTOMER

	# Distinct values	Min Value	Max Value
C_CUSTKEY	1,500,000	1	1,500,000
C_NAME	1,500,000	Customer #000000001	Customer #001500000
C_NATIONKEY	25	0	24

SUPPLIER

	# Distinct values	Min Value	Max Value
S_SUPPKEY	100,000	1	100,000
S_NATIONKEY	25	0	24

REGION

	# Distinct values	Min Value	Max Value
R_REGIONKEY	5	0	4
R_NAME	5	AFRICA	MIDDLE EAST

NATION

	# Distinct values	Min Value	Max Value
N_NATIONKEY	25	0	24
N_NAME	25	ALGERIA	VIETNAM
N_REGIONKEY	5	0	4

4 Definition of the set of queries

As already mentioned, we worked on query schemata 1 and 3.

4.1 Query schemata 1

The first query schemata returned an aggregation of the export/import revenue from lineitems between exporting and importing nations.
The revenue was obtained as l_extendedprice * (1 - l_discount) of the considered lineitems.
The aggregation was performed using the following roll-up:

- Month → Quarter → Year
- Type
- Nation → Region

The slicing was over Type and Exporting Nation.

Below the query that we implemented:

```sql
SELECT
    ExpReg.r_name AS export_region,
    ImpReg.r_name AS import_region,
    ExpNat.n_name AS export_nation,
    ImpNat.n_name AS import_nation,
    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 Ex ON Ex.s_suppkey = L.l_suppkey
    JOIN CUSTOMER AS Im ON Im.c_custkey = O.o_custkey
    JOIN NATION AS ExpNat ON ExpNat.n_nationkey = Ex.s_nationkey
    JOIN NATION AS ImpNat ON ImpNat.n_nationkey = Im.c_nationkey
    JOIN REGION AS ExpReg ON ExpReg.r_regionkey = ExpNat.n_regionkey
```

```
    JOIN REGION AS ImpReg ON ImpReg.r_regionkey = ImpNat.n_regionkey
WHERE
    ExpNat.n_name = 'FRANCE'
    AND ImpNat.n_name != ExpNat.n_name
    AND P.p_type = 'SMALL POLISHED TIN'
GROUP BY
    ROLLUP(P.p_type),
    ROLLUP(ExpReg.r_name, ExpNat.n_name),
    ROLLUP(ImpReg.r_name, ImpNat.n_name),
    ROLLUP(DATE_PART('year', O.o_orderdate),
           DATE_PART('quarter', O.o_orderdate),
           DATE_PART('month',O.o_orderdate));
```

4.2 Query schemata 3

The third query schemata returned the revenue loss for customers who might be having problems with the parts that are shipped to them. Revenue loss was defined as the sum of l_extendedprice * (1 - l_discount) for all qualifying lineitems.
The aggregation was performed using the following roll-up:

- Month → Quarter → Year
- Customer

The query was issued with slicing on the name of a customer.

Below the query that we implemented:

```
SELECT
  COALESCE(DATE_PART('year', O.o_orderdate)::text, 'Total') AS yearOrder,
  DATE_PART('quarter', O.o_orderdate) AS quarterOrder,
  DATE_PART('month', O.o_orderdate) AS monthOrder,
  CU.c_name AS custName,
  SUM(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
  customer AS CU ON CU.c_custkey = O.o_custkey
WHERE
  L.l_returnflag = 'R'
  AND CU.c_name ='Customer#000002000'
GROUP BY
    ROLLUP(yearOrder, quarterOrder, monthOrder),
    ROLLUP(custName)
ORDER BY
  custName,
  COALESCE(DATE_PART('year', O.o_orderdate)::text, 'All Years'),
  quarterOrder,
  monthOrder;
```

5 Query cost before optimization

As already mentioned before, to evaluate the performance of the queries we used the `time` function within the psql tool and we executed each query a total of 30 times. Below we find the mean and the standard deviation of the real and the CPU time taken by the non-optimized queries.

	Real time		CPU time	
	μ	σ	μ	σ
Query 1	1.597	0.011	0.082	0.002
Query 3	0.026	0.002	0.021	0.002

Figure 1: Execution time with no optimization (in sec.)

6 Optimization with Indexes

To optimize our database queries, we initially indexed all attributes involved in JOIN operations and WHERE conditions, anticipating that these would impact performance the most.
We then used the Explain Analyze command to observe which indexes the query execution plan (QEP) actually utilized. Based on this analysis, we retained only those indexes proven to be effective, streamlining our database's performance by aligning our indexing strategy directly with the optimizer's actual usage patterns.

The SQL code for defining the indexes follows.

```sql
CREATE INDEX idx_customer_cname ON customer(c_name);
CREATE INDEX idx_lineitem_lorderkey ON lineitem(l_orderkey);
CREATE INDEX idx_lineitem_lpartkey ON lineitem(l_partkey);
CREATE INDEX idx_orders_ccustkey ON orders(o_custkey);
CREATE INDEX idx_part_ptype ON part(p_type);
CREATE INDEX idx_supplier_snationkey ON supplier(s_nationkey);
```

The following table provides an overview of our indexing strategy, detailing the targeted attributes, their associated tables, the incurred space overhead for maintaining these indexes, and their direct utility in optimizing specific queries.

Attribute	Table	Space	Usage
c_name	Customer	52.12 MB	Q3
l_orderkey	Lineitem	742.38 MB	Q3
l_partkey	Lineitem	429.51 MB	Q1
o_custkey	Orders	120.24 MB	Q3
p_type	Part	13.66 MB	Q1
s_nationkey	Supplier	704 KB	Q1
TOTAL		**1.36 GB**	

Figure 2: Selected indexes

Indexing has resulted in an additional memory usage of 1.36 GB, for a total of 16.96 GB space needed, which is within the space constraint of using at most 1.5 times the size of the database.

6.1 Results of optimization with indexes

In Figure 3 we can see how indexing has affected performance:

		Real time		CPU time	
		μ	σ	μ	σ
Query 1	No optimization	2.665	0.113	0.082	0.004
	With indexes	**1.597**	0.011	**0.082**	0.002
Query 3	No optimization	0.344	0.004	0.022	0.001
	With indexes	**0.026**	0.002	**0.021**	0.002

Figure 3: Execution time with indexes (in sec.)

We observe that real time execution significantly drops for both queries, while CPU time slightly improves for query 3 and stays constant for query 1.

7 Optimization with materialization

In order to study the effects of materialized views, we firstly removed the indexes that we had added in the previous optimization step and then created the materialized views.

To decide which materialized view to use, we started by observing which were the tables and the table attributes used by the two queries.
We initially created a unique materialized view which would join all the tables used by the two queries and project all and only the attributes needed by the queries. Nevertheless, its total required space was over 12 GB so we drop it since it was not respecting the constraint set by the exercise.
We then kept trying and we report below the procedure and the results of the two final attempts that we made.

7.1 Attempt A

Instead of using a single big materialized view, we decided to create multiple small ones:

- *lineitem_orders_customer*: it joins the three tables of Lineitem, Orders and Customer and it is used to run both query schemata, the one on export/import revenue value, and the one on the returned item loss.
- *supplier_info* and *customer_info*: they respectively join the tables Supplier and Customer with the tables Nation and Region, and they are used for the first query schema only.

The three materialized views are presented below:

```
CREATE MATERIALIZED VIEW lineitem_orders_customer AS
SELECT
    orders.o_orderkey,
    orders.o_orderdate,
    orders.o_custkey,
    (lineitem.l_extendedprice * (1 - lineitem.l_discount)) AS revenue
    lineitem.l_returnflag,
    lineitem.l_partkey,
    lineitem.l_suppkey,
    customer.c_name
    FROM lineitem
    JOIN orders ON lineitem.l_orderkey = orders.o_orderkey
    JOIN customer ON customer.c_custkey = orders.o_custkey;
```

```
CREATE MATERIALIZED VIEW supplier_info AS
 SELECT
    supplier.s_suppkey,
    supplier.s_name,
    nation.n_nationkey AS s_nationkey,
    nation.n_name AS s_nationname,
    region.r_regionkey AS s_regionkey,
    region.r_name AS s_regionname
    FROM supplier
    JOIN nation ON supplier.s_nationkey = nation.n_nationkey
    JOIN region ON nation.n_regionkey = region.r_regionkey;
```

```
CREATE MATERIALIZED VIEW customer_info AS
 SELECT customer.c_custkey,
    customer.c_name,
```

```
    nation.n_nationkey AS c_nationkey,
    nation.n_name AS c_nationname,
    region.r_regionkey AS c_regionkey,
    region.r_name AS c_regionname
    FROM customer
    JOIN nation ON customer.c_nationkey = nation.n_nationkey
    JOIN region ON nation.n_regionkey = region.r_regionkey;
```

The space taken by the three materialized view is indicated in Figure 4

	SPACE	USAGE
lineitem_orders_customer	4.72 GB	Q1, Q3
supplier_info	12.02MB	Q1
customer_info	167.41MB	Q1
TOTAL	**4.80GB**	

Figure 4: Space of the materialized views - attempt A

Based on the materialized views that we created, we edited the two queries as indicated below:

Query 1

```
SELECT
  s_regionname AS export_region,
  c_regionname AS import_region,
  s_nationname AS export_nation,
  c_nationname AS import_nation,
  SUM(revenue) AS revenue,
  DATE_PART('month', o_orderdate) AS order_month,
  DATE_PART('quarter', o_orderdate) AS order_quarter,
  DATE_PART('year', o_orderdate) AS order_year,
  p_type AS ptype
FROM
    lineitem_orders_customer
    JOIN PART ON p_partkey = l_partkey
    JOIN supplier_info ON s_suppkey=l_suppkey
    JOIN customer_info ON c_custkey=o_custkey
WHERE
    s_nationname = 'FRANCE'
    AND c_nationname != s_nationname
    AND p_type = 'SMALL␣POLISHED␣TIN'
GROUP BY
    ROLLUP(p_type),
    ROLLUP(s_regionname, s_nationname),
    ROLLUP(c_regionname, c_nationname),
    ROLLUP(DATE_PART('year', o_orderdate),
           DATE_PART('quarter', o_orderdate),
           DATE_PART('month', o_orderdate));
```

Query 3

```
SELECT
  DATE_PART('year', o_orderdate) AS yearOrder,
  DATE_PART('quarter', o_orderdate) AS quarterOrder,
  DATE_PART('month', o_orderdate) AS monthOrder,
  c_name AS custName,
  SUM(revenue) AS revenue
FROM
  lineitem_orders_customer
WHERE
  l_returnflag = 'R'
  AND c_name ='Customer#000002000'
GROUP BY
  ROLLUP(custName),
  ROLLUP(DATE_PART('year', o_orderdate),
         DATE_PART('quarter', o_orderdate),
         DATE_PART('month', o_orderdate));
ORDER BY
 custName,
 COALESCE(DATE_PART('year', o_orderdate)::text, 'All Years'),
 quarterOrder,
 monthOrder;
```

7.1.1 Results of optimization with materialized views - attempt A

In Figure 5, we can see that, for both queries, the total execution time increased when using the materialized views compared to the original situation of no optimization, while the CPU time decreased.

		Real time		CPU time	
		μ	σ	μ	σ
Query 1	No optimization	2.665	0.113	0.082	0.004
	With materialized views	**2.942**	0.022	**0.073**	0.001
Query 3	No optimization	0.344	0.004	0.022	0.001
	With materialized views	**1.045**	0.016	**0.021**	0.001

Figure 5: Execution time with materialized views - attempt A (in sec.)

7.2 Attempt B

Given that the previous attempt led to a total execution time higher than the time needed when no optimization technique was considered, we tried another approach. Instead of creating materialized views that could be applicable to both queries (as it was the materialized view *lineitem_orders_customer*), we decided to create materialized views focused on each individual query.

For the input/output revenue query, we created the materialized view below:

```sql
CREATE MATERIALIZED VIEW mv_q1 AS
SELECT
    ExpReg.r_regionkey AS export_region,
    ImpReg.r_regionkey AS import_region,
    ExpNat.n_nationkey AS export_nation,
    ImpNat.n_nationkey AS import_nation,
    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 Ex ON Ex.s_suppkey = L.l_suppkey
    JOIN CUSTOMER AS Im ON Im.c_custkey = O.o_custkey
    JOIN NATION AS ExpNat ON ExpNat.n_nationkey = Ex.s_nationkey
    JOIN NATION AS ImpNat ON ImpNat.n_nationkey = Im.c_nationkey
    JOIN REGION AS ExpReg ON ExpReg.r_regionkey = ExpNat.n_regionkey
    JOIN REGION AS ImpReg ON ImpReg.r_regionkey = ImpNat.n_regionkey
WHERE
    ImpNat.n_nationkey != ExpNat.n_nationkey
```

For the query on the revenue loss, we created the materialized view below:

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

For query 1, in order to minimize the space occupied by the materialized view, we decided to use `r_regionkey` and `n_nationkey` instead of `r_name` and `n_name`. This choice allowed us to spare approximately 5 GB of memory. Indeed, as we can see from the tables' definitions, `r_name` and `n_name` are of type CHAR(25) which means each entry weighs 25 bytes, whereas `r_regionkey` and `n_nationkey` are integers (4 bytes in most systems). The space taken by the two materialized views is indicated in Figure 6. Even though it is a bit larger than the space taken with the previous attempt, (since here both materialized views use the Lineitem table, which is the largest one of the dataset), we are still below the space constraint set by the exercise.

	SPACE	USAGE
mv_q1	4.46 GB	Q1
mv_q3	1.16 GB	Q3
TOTAL	**5.62GB**	

Figure 6: Space of the materialized views - attempt B

Based on the new created materialized views, we edited the two queries as reported below:

Query 1

```
SELECT
    DATE_PART('year', orderdate) AS order_year,
    DATE_PART('quarter', orderdate) AS order_quarter,
    DATE_PART('month', orderdate) AS order_month,
    ExpNat.n_name AS exp_nation,
    ImpNat.n_name AS imp_nation,
    ExpReg.r_name AS exp_region,
    ImpReg.r_name AS imp_region,
    SUM(revenue) AS total_revenue,
    ptype
FROM
    mv_q1
    JOIN nation AS ExpNat ON ExpNat.n_nationkey=export_nation
    JOIN nation AS ImpNat ON ImpNat.n_nationkey=import_nation
    JOIN region AS ExpReg ON ExpReg.r_regionkey=export_region
    JOIN region AS ImpReg ON ImpReg.r_regionkey=import_region
WHERE
    ExpNat.n_name = 'FRANCE'
    AND ptype = 'SMALL POLISHED TIN'
GROUP BY
    ROLLUP(ptype),
    ROLLUP(exp_region, exp_nation),
    ROLLUP(imp_region, imp_nation),
    ROLLUP(order_year, order_quarter, order_month);
```

Query 3

```
SELECT
  DATE_PART('year', o_orderdate) AS yearOrder,
  DATE_PART('quarter', o_orderdate) AS quarterOrder,
  DATE_PART('month', o_orderdate) AS monthOrder,
  c_name AS custName,
  SUM(revenue) AS total_revenue
FROM
  mv_q3
WHERE
  c_name ='Customer#000002000'
GROUP BY
```

```
  ROLLUP (custName),
  ROLLUP (yearOrder, quarterOrder, monthOrder)
ORDER BY
 custName,
 COALESCE(DATE_PART('year', o_orderdate)::text, 'All Years'),
 quarterOrder,
 monthOrder;
```

7.2.1 Results of optimization with materialized views - attempt B

In Figure 7 we can see that, for both queries, the time needed for the whole execution by using the materialized views is now less than the time needed in a situation with no optimization. This may be due to how we designed the materialized views: the size of the tables that we join in query 1 decreases compared to attempt A, and the `WHERE` condition on `l_returnflag` significantly helps for query 3.

		Real time		CPU time	
		μ	σ	μ	σ
Query 1	No optimization	2.665	0.113	0.082	0.004
	With materialized views	**1.633**	0.028	**0.075**	0.001
Query 3	No optimization	0.344	0.004	0.022	0.001
	With materialized views	**0.279**	0.106	**0.022**	0.002

Figure 7: Execution time with materialized views - attempt B (in sec.)

8 Optimization with indexes and materialized views

When constructing the indexes on the materialized views, we decided to construct them on both attempt A and B of the materialized views described above. For both cases, we proceeded as done in section 6, so we started by identifying all the attributes that are used in the JOIN operations and in the WHERE clauses of our two queries and we created an index for each one of them.

8.1 Indexes on materialized views - attempt A

For the first attempt, we started creating indexes on eight attributes and then kept only the five of them that were actually used based on the Explain Analyze check:

Attribute	Table or materialized view	Space	Usage
l_partkey	MV lineitem_orders_customer	429.51 MB	Q1
c_name	MV lineitem_orders_customer	419.69 MB	Q3
s_nationname	MV supplier_info	728 KB	Q1
c_custkey	MV customer_info	32.16 MB	Q1
p_type	Part	13.66 MB	Q1
TOTAL		**0.895 GB**	

Figure 8: Selected indexes on materialized views - attempt A

8.1.1 Results of optimization with indexes and materialized views - attempt A

When executing the queries using the indexes created on the initial materialized views, we noticed that the total time needed beneficiated from the presence of the indexes compared to the situation with no optimization, even though the use of those same materialized views before the indexing had led to a worse performance in terms of time needed.

		Real time		CPU time	
		μ	σ	μ	σ
Query 1	No optimization	2.665	0.113	0.082	0.004
	Indexes & mat. views	**1.769**	0.012	**0.073**	0.002
Query 3	No optimization	0.344	0.004	0.022	0.001
	Indexes & mat. views	**0.025**	0.003	**0.021**	0.002

Figure 9: Execution time indexes and materialized views - attempt A (in sec.)

8.2 Indexes on materialized views - attempt B

For the second attempt, we identified six potentially useful attributes and then kept only the three of them that were actually used when executing the queries:

Attribute	Table or materialized view	Space	Usage
ptype	mv_q1	390.88 MB	Q1
export_nation	mv_q1	380.62 MB	Q1
c_name	mv_q3	136.16 MB	Q3
TOTAL		**0.907 GB**	

Figure 10: Selected indexes on materialized views - attempt B

8.2.1 Results of optimization with indexes and materialized views - attempt B

In Figure 11, we can see that the combined use of indexes and materialized views led to an improvement in terms of total execution time compared to the situation with no optimization. Nevertheless, for query 1, indexing slightly worsens the performance with respect to the materialized view with no indexes. Indexing, in fact, brings some overhead that can become more relevant than its advantages when the selectivity rate of the indexed attributes is low (it's the case of `ptype` and `export_nation` of `mv_q1`).

		Real time		CPU time	
		μ	σ	μ	σ
Query 1	No optimization	2.665	0.113	0.082	0.004
	Indexes & mat. views	**1.942**	0.007	**0.075**	0.002
Query 3	No optimization	0.344	0.004	0.022	0.001
	Indexes & mat. views	**0.026**	0.002	**0.022**	0.001

Figure 11: Execution time indexes and materialized views - attempt B (in sec.)

9 Recap of the optimization strategies

Our initial database takes 15.6 GB so, with the use of any optimization technique, we needed to stay within 23.4 GB to respect the space constraint set by the exercise.
The table below presents a recap of all the above mentioned optimization solutions.

	Extra space (in GB)	Total space (in GB)	Execution time (in sec.)	
			Query 1	Query 3
No optimization		15.6	2.665	0.344
Indexes	1.36	16.96	1.597	0.026
Materialized views - A	4.8	20.4	2.942	1.045
Materialized views - B	5.62	21.22	1.633	0.279
Indexes+materialized views - A	4.80 + 0.89	21.29	1.769	0.025
Indexes+materialized views - B	5.62 + 0.91	22.13	1.942	0.026

Figure 12: Recap of all optimization techniques

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

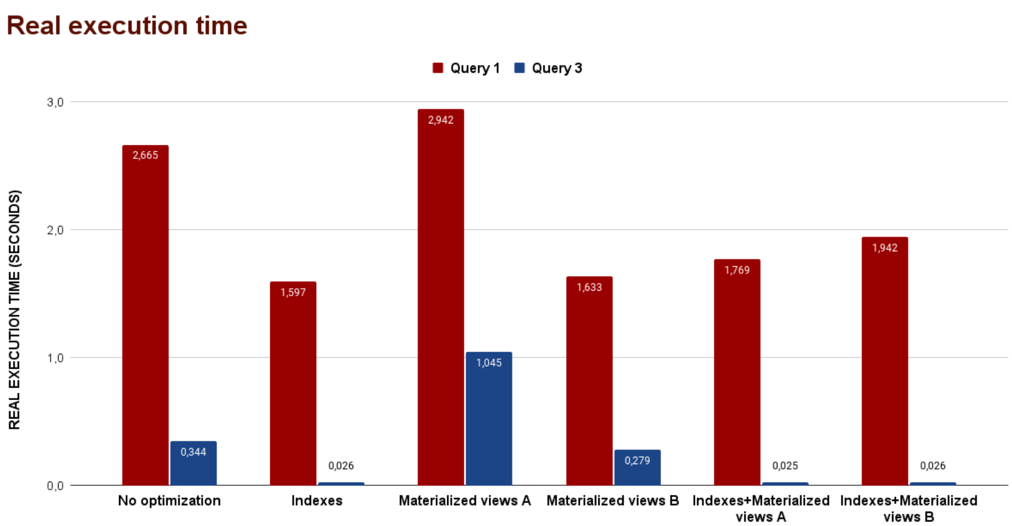

Figure 13: Recap of all optimization techniques (real execution time)

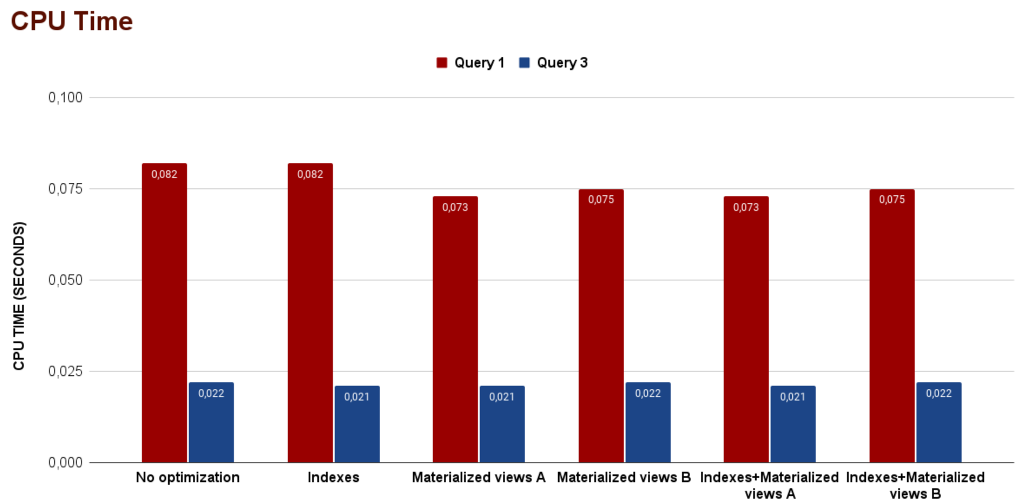

Figure 14: Recap of all optimization techniques (CPU execution time)

10 Final Considerations and conclusions

The strategy yielding the best results, both in terms of space cost and execution time, consisted in using appropriate indexes on the original tables.
Database optimization involves finding the right balance between using system resources efficiently and improving performance. Our research explored different strategies, including adding complexity like materialized views and indexes to the database schema. Not always adding complexity (like in the case of "materialized view B with indexes") led to better results: the overhead introduced both by indexes and materialized views must always be taken into account.
When dealing with materialized views our fundamental aim was to minimize the space they occupied while maximizing the operations they precomputed for the queries. When focusing on the indexes instead, we noticed how their positive impact was actually significant when they are defined on attributes with high selectivity rate.
In any case, a thorough knowledge of the data that we are working with is crucial to design the most suitable solutions.