Data Management - Final Project

Sara Carpenè, Alessio Valentinis, Marco Zampar

July 22, 2024

1 Introduction

This project has the aim of optimizing a set of four queries from the TPC-Benchmark H database. The database consists of eight tables: customer, lineitem, nation, orders, part, partsupp, region, supplier. The relations between tables can be seen in the schema in figure 1.

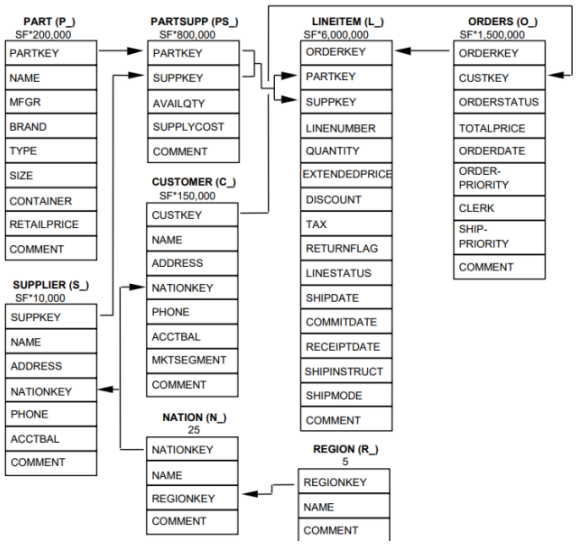

Figure 1: Schema of the relation between tables in the TPC-H benchmark

1.1 Creation and population of the database

The size of the database is scalable, depends on a scale factor (SF) and we were given data generated from $SF = 10$. Each table, from the documentation, has its own primary key and one or more foreign keys. Complete description of the table can be found here, while the complete SQL tables implementation and creation can be found here.

For the purpose of space economy, we opted for an initial "vanilla" creation of the tables, without any kind of keys: this choice is furthermore supported by the fact that we are dealing

with a Data Warehousing context, so we expect the data to be already prepared and cleaned up from duplicates.

After having populated the tables, we inserted all the keys reported in the documentation in order to work properly with the relations. However, always for the purpose of gaining some space, we decided not to implement the *Primary key (l_orderkey, l_linenumber)* to the fact table *lineitem*, as, for the same reasons described above, we don't need to check for uniqueness constraints during the population of the database, and for the sake of optimization, a simple index should help us to spare some pretty useful space.

In the table 1, we provide some information about the dimension of the various tables, in increasing order.

Table	Number of rows	Dim without keys (MB)	Dim with keys (MB)
region	5	0.01	0.02
nation	25	0.01	0.02
supplier	100000	17.35	19.51
customer	1500000	290.17	322.32
part	2000000	320.14	363.00
partsupp	8000000	1362.80	1535.12
orders	15000000	2038.97	2360.30
lineitem	59986052	8787.95	10073.67

Table 1: General statistics

1.2 Measurement techniques

In order to assess the execution times of the various queries, we used the `EXPLAIN ANALYZE` feature of *PostgreSQL*. It provides a complete summary of the execution plan of the queries, together with execution time in milliseconds. In order to have some significance on the result, we executed every query 5 times, and then we computed the mean and standard deviation of the measurements.

In order to keep track of the dimension of the database, we used the `SELECT pg_database_size();` command, and convert the dimension in MB or GB.

1.3 Hardware specifications

All the test were conducted on a MacBook Air laptop with the following characteristics:

- CPU: Chip Apple M2, 8 core (4 performance cores and 4 efficiency cores);
- RAM: 8GB;
- SSD: SATA 256GB;
- GPU: built-in 10 cores GPU;
- OS: macOS Sonoma 14.2.1.

2 Statistics of the DB

In this section, we will present some useful statistics of the database.

The original database has a total dimension of $14.34GB$, this encompasses both the physical size of the tables and the dimensions of primary and foreign keys.

In appendix A more complete statistics of the tables can be found, encompassing all their attributes along with the count of unique values, minimum, and maximum values for each attribute.

Here, to keep the focus on the four queries that we want to optimize, we will limit the presentation to some of those statistics. Specifically we decided to mention only the tables and the attributes that will be involved in at least one of the chosen query.

Attribute	Distinct values	Min value	Max value
c_custkey	1500000	1	1500000
c_name	1500000	'Customer#000000001'	'Customer#001500000'
c_address	1500000	-	-
c_nationkey	25	0	24
c_phone	1499963	-	-
c_acctbal	818834	-999.99	9999.99
c_comment	1496636	-	-

Table 2: Costumer statistics

Attribute	Distinct values	Min value	Max value
p_partkey	2000000	1	2000000
p_type	150	-	-
p_container	40	-	-

Table 3: Part statistics

Attribute	Distinct values	Min value	Max value
l_orderkey	15000000	1	60000000
l_partkey	2000000	1	2000000
l_quantity	50	1	50
l_extendedprice	1351462	900.91	104949.5
l_discount	11	0.0	0.1
l_tax	9	0.0	0.08
l_returnflag	3	-	-
l_linestatus	2	-	-
l_shipdate	2526	'1992-01-02'	'1998-12-01'
l_linenumber	7	1	7

Table 4: Lineitem statistics

Attribute	Distinct values	Min value	Max value
o_orderkey	15000000	1	60000000
o_custkey	999982	1	1499999
o_orderdate	2406	'1992-01-01'	'1998-08-02'

Table 5: Orders statistics

Attribute	Distinct values	Min value	Max value
n_nationkey	25	0	24
n_name	25	-	-

Table 6: Nation statistics

3 Query schemas

The assignment consists in using TPC-Benchmark H to test and optimize a set of four queries, using indexes, materialized views, a mixed approach of the two and fragmentation.

The set of queries selected for the assignment are Q1, Q10, Q14, Q17 of the Official Documentation. An overall view on the description and SQL implementation is given below.

3.1 Query 1

Brief description The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60 - 120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included.

Functional definition

```
SELECT
    l_returnflag,
    l_linestatus,
    SUM(l_quantity) AS sum_qty,
    SUM(l_extendedprice) AS sum_base_price,
    SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    AVG(l_quantity) AS avg_qty,
    AVG(l_extendedprice) AS avg_price,
    AVG(l_discount) AS avg_disc,
    COUNT(*) AS count_order
FROM
    lineitem
WHERE
```

```
    l_shipdate <= DATE '1998-12-01' - INTERVAL '[DELTA]' DAY
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

For a matter of simplicity we decided to take as slicing values the ones proposed by the validation paragraph in the official documentation. (So `'[DELTA]' = '90'`)

3.2 Query 10

Brief description The Returned Item Reporting Query finds the top 20 customers, in terms of their effect on lost revenue for a given quarter, who have returned parts. The query considers only parts that were ordered in the specified quarter. The query lists the customer's name, address, nation, phone number, account balance, comment information and revenue lost. The customers are listed in descending order of lost revenue. Revenue lost is defined as sum(l_extendedprice*(1-l_discount)) for all qualifying lineitems.
Functional definition

```
SELECT
    c_custkey,
    c_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
FROM
    customer,
    orders,
    lineitem,
    nation
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate >= DATE '[DATE]'
    AND o_orderdate < DATE '[DATE]' + INTERVAL '3' MONTH
    AND l_returnflag = 'R'
    AND c_nationkey = n_nationkey
GROUP BY
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
```

```
ORDER BY
    revenue DESC;
```

For a matter of simplicity we decided to take as slicing values the ones proposed by the validation paragraph in the official documentation. (So `'[DATE]' = '1993-10-01'`)

3.3 Query 14

Brief description The Promotion Effect Query determines what percentage of the revenue in a given year and month was derived from promotional parts. The query considers only parts actually shipped in that month and gives the percentage. Revenue is defined as (l_extendedprice * (1-l_discount)).
Functional definition

```
SELECT
    100.00 * SUM (CASE WHEN p_type like 'PROMO%'
                  THEN l_extendedprice*(1-l_discount)
                  ELSE 0 END) / SUM(l_extendedprice * (1 - l_discount))
    AS promo_revenue
FROM
    lineitem,
    part
WHERE
    l_partkey = p_partkey
    AND l_shipdate >= date '[DATE]'
    AND l_shipdate < date '[DATE]' + interval '1' month;
```

For a matter of simplicity we decided to take as slicing values the ones proposed by the validation paragraph in the official documentation. (So `'[DATE]' = '1995-09-01'`)

3.4 Query 17

Brief description The Small-Quantity-Order Revenue Query considers parts of a given brand and with a given container type and determines the average lineitem quantity of such parts ordered for all orders (past and pending) in the 7-year database. What would be the average yearly gross (undiscounted) loss in revenue if orders for these parts with a quantity of less than 20 % of this average were no longer taken?
Functional definition

```
SELECT
    SUM(l_extendedprice) / 7.0 AS avg_yearly
FROM
    lineitem,
    part
WHERE
    p_partkey = l_partkey
    AND p_brand = '[BRAND]'
    AND p_container = '[CONTAINER]'
```

```
    AND l_quantity < (
                        SELECT
                            0.2 * AVG(l_quantity)
                        FROM
                            lineitem
                        WHERE
                            l_partkey = p_partkey
                      );
```

For a matter of simplicity we decided to take as slicing values the ones proposed by the validation paragraph in the official documentation.

(So `'[BRAND]' = 'Brand#23'` and `'[CONTAINER]' = 'MED BOX'`)

4 Baseline

We decided to test the execution time of queries without additional indexes or views, other than the primary keys suggested by the documentation, and to use it as a baseline for further improvement in the management of the queries.

In order to keep the results as similar as possible to the theoretical case, we decided to disable any kind of hash-related operations, that in PostgreSQL are very much used and optimized, and can alter the improvement of the execution time when introducing indexes.

The results of the tests are reported in the table 7.

The execution times for Query 17 are not reported in the table because it was not feasible to execute it within reasonable time frames to obtain mean and standard deviation values. Since after two hours of computation it had still not produced a result, we decided to estimate empirically the possible the execution times. We ran the query on a limited dataset, attempting to understand the relationship between the number of rows in the table and execution time. Given that executing the query on data related to one month, or 103.000 rows, requires a cost of 144.035 operations and takes 2 seconds, and running the query on data related to three months, or 8.814 rows, requires a cost of 6.091.570 and takes 173 seconds, we reasonably assumed that executing the query on the entire table composed of 60 million rows, with a cost of 117.789.668.871 operations, would take significantly longer than what we had available. Additionally, since the purpose of data warehousing is to support the decision-making, this query without optimization becomes useless and thus reporting its cost would be irrelevant if it exceeds a couple of hours.

Query	Mean [s]	Std [s]
Q1	41.739	1.412
Q10	33.077	1.634
Q14	28.253	1.239
Q17	N/A	N/A

Table 7: Execution times of query for baseline

5 Indexes

Our first attempt was to add indexes on foreign keys, but in almost all cases, they weren't used, or didn't bring too many advantages, compared with their size.

Our second attempt was to add indexes on the attributes used for slicing, so involved in the `WHERE` condition.

Table	Attribute	Used in Query	Creation time [s]	Index size [MB]
lineitem	l_shipdate	Q1 and Q14	32.43	397.54
lineitem	l_returnflag	Q10	61.83	396.46
lineitem	l_partkey	Q17	46.99	429.50

Table 8: Indexes dimensions

With these indexes, which are ensured to be used in the execution of the queries, resulted in a total database size of $15.32GB$.

- Query 1: even if the condition `l_shipdate <= DATE '1998-12-01' - INTERVAL '90' DAY` is not very selective, from the table below we can see that in the index on *l_shipdate* in any case helpful.

  We also tried to optimise the `GROUP BY` of Query 1 with an index on *(l_returnflag, l_linestatus)* or on *l_returnflag, l_linestatus* separately, but they were not used.

- Query 10: The selectivity of `l_returnflag = R` is not high (~ 0.25), anyway improves the execution time.

  We also tried to introduce an index over *o_orderdate* to optimize query 10 but, analyzing the execution plan, it was not used. If we force the optimizer to actually use it, we obtain worse performances, so we decided to drop it.

- Query14: the index on *l_shipdate* is very selective and no other indexes were useful beacuse a `MERGE JOIN` is performed.

- Query 17: the index on *l_partkey* is fundamental, because of the way in which the query is written: for every line of `part` table, we should scan the entire `lineitem` table, and check for the condition `p_partkey=l_partkey`.

  We tried an index on *p_brand* and *p_container* but they were not included in the execution plan by the optimizer.

  It is interesting to note that by simply adding an index on *l_partkey* we made this query feasible.

The execution time of each query is summarized in the table below.

Query	Mean [s]	Std [s]
Q1	30.009	1.224
Q10	25.677	1.799
Q14	23.607	0.327
Q17	11.232	0.967

Table 9: Execution times of query with indexes

6 Materialized views

In this section we will propose some materialized views that aim to improve the execution time of the chosen queries. For the creation of this views we enabled all the hash related operations, since the main purpose of the section is to evaluate performances related to materialization of some tables, rather than delving into the specifics of this process.

6.1 Lineitem-part

In order to improve performances in executing query 14 we decided to create a materialized view as follow:

```
CREATE MATERIALIZED VIEW part_lineitem AS
SELECT
    l_returnflag,
    l_linestatus,
    l_quantity,
    l_extendedprice,
    l_discount,
    l_tax,
    l_shipdate,
    l_partkey,
    p_partkey,
    p_brand,
    p_container,
    SUBSTRING(p_type FROM 1 FOR 5) AS p_type_prefix,
    0.2 * AVG(l_quantity) OVER (PARTITION BY l_partkey) AS avg_quantity
FROM
    lineitem l
JOIN
    part p ON l.l_partkey = p.p_partkey;
```

Statistics of this view:

- Required time to create the view 310.952 seconds.
- Size of the view: 6.43 GB.

We rewrote the query 14 and 17 in order to exploit the materialized views.

Materializing the join operation of query 14 we expected to have a lower execution time with respect to the baseline, but this did not happen. By observing the output of the

EXPLAIN ANALYZE function we understood that the problem with the lack of gain in performance is that the optimizer performs a sequential scan of the *lineitem* table, since there is no index on *l_shipdate*.

The greatest improvement in using this materialization was expected to be in query 17. Indeed the execution time dropped to 15.265 seconds (on a single run), which is a sensible gain in terms of performances.

6.2 Customer-orders-lineitem-nation

In order to spare the most time of the joins in query 10 we created a bigger materialized view.

```
CREATE MATERIALIZED VIEW customer_order_lineitem_nation AS
SELECT
    c.c_custkey,
    c.c_name,
    c.c_acctbal,
    n.n_name,
    c.c_address,
    c.c_phone,
    c.c_comment,
    l.l_returnflag,
    l.l_discount,
    l.l_extendedprice,
    o.o_orderdate

FROM
    customer c
JOIN
    orders o ON c.c_custkey = o.o_custkey
JOIN
    lineitem l ON l.l_orderkey = o.o_orderkey
JOIN
    nation n ON c.c_nationkey = n.n_nationkey;
```

Statistics of this view:

- Required time to create the view 437.107 seconds.
- Size of the view: 12.63 GB.

We tested the performances of query 10 rewritten using this materialization, but there was no gain in efficiency, other than the fact that this materialization was made ad-hoc for this single query.

6.3 Lineitem-part-orders

As a last option we opted for a mixed approach creating a view which is neither as big nor as specific as the previous ones.

```sql
CREATE MATERIALIZED VIEW part_lineitem_order AS
SELECT
    l_returnflag,
    l_linestatus,
    l_quantity,
    l_extendedprice,
    l_discount,
    l_tax,
    l_shipdate,
    l_partkey,
    p_partkey,
    p_brand,
    p_container,
    SUBSTRING(p_type FROM 1 FOR 5) AS p_type_prefix,
    0.2 * AVG(l_quantity) OVER (PARTITION BY l_partkey) AS avg_quantity,
    o_orderkey,
    o.o_custkey,
    o.o_orderdate
FROM
    lineitem l
JOIN
    part p ON l.l_partkey = p.p_partkey
JOIN
    orders o ON l.l_orderkey = o.o_orderkey;
```

Statistics of this view:

- Required time to create the view 366.508 seconds.
- Size of the view: 6.93 GB.

The total database size is 21.27 GB.
As this is the expected best materialization, we report the modified queries that use it.

Query 10

```sql
SELECT
    c_custkey,
    c_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
FROM
    part_lineitem_order
    JOIN customer c ON c.c_custkey = o_custkey
    JOIN nation n ON c.c_nationkey = n.n_nationkey
WHERE
    o_orderdate >= DATE '1993-10-01'
    AND o_orderdate < DATE '1993-10-01' + INTERVAL '3' MONTH
```

```
    AND l_returnflag = 'R'
GROUP BY
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
ORDER BY
    revenue DESC;
```

Query 14

```
SELECT
    100.00 * SUM(CASE
        WHEN p_type_prefix LIKE 'PROMO'
        THEN l_extendedprice * (1 - l_discount)
        ELSE 0
    END) / SUM(l_extendedprice * (1 - l_discount)) AS promo_revenue
FROM
    part_lineitem_order
WHERE
    l_shipdate >= DATE '1995-09-01'
    AND l_shipdate < DATE '1995-09-01' + INTERVAL '1' MONTH;
```

Query 17

```
SELECT
    SUM(l_extendedprice) / 7.0 AS avg_yearly
FROM
    part_lineitem_order
WHERE
    p_brand = 'Brand#23'
    AND p_container = 'MED BOX'
    AND l_quantity < avg_quantity;
```

In this case we performed a complete benchmark, and the results are reported in the table below 10.

Query	Mean	Std
Q1	41.739	1.412
Q10	24.535	2.347
Q14	21.885	1.037
Q17	20.465	0.655

Table 10: Execution times of query with materialized views

As we can see from the table above, the execution time for query 1 is not changed from the baseline, this was expected since the introduced materialization is not involved in this case.

For the sake of completeness, we will briefly present an idea to optimize with materialization also query one, which is the only query in the chosen set that isn't directly influenced by the already introduced materialized views. We proposed a materialization that returns a "slimmer" version of the *lineitem* table with less attributes than the original one. Specifically we decided to keep only *l_returnflag, l_linestatus, l_extendedprice,l_discount, l_tax, l_quantity.*

This approach leaded to a slight decrease in the execution time of query 1, but this improvement was not so significant to balance the size of the materialized view and its specificity. Indeed this materialization is "tailored" to query 1 and it could hardly be used in the optimization of other queries.

Since, in completing this project, we have always considered our set of four queries as a small subset of the possible query that a company may see as interesting in the decision making process, we didn't want to burden the database with too specific materialized views. For this reason we decided to not consider also this materialization, and to continue the project with only the materialized view that involves *lineitem*, *part* and *orders* tables.

Let's see how many execution of every query are needed to make the usage of the materialized view convenient.

we want to find the smallest n for which creating the materialization and executing the query n times with it require less time than directly execute n times the query without optimization.

- Query 1: Since the times are equal to the one on the baseline, it doesn't make sense to compute this quantity.
- Query 10:
$$n \cdot 33.077 > n \cdot 24.535 + 366.508$$
Which means $n \geq 23$.
- Query 14:
$$n \cdot 28.253 > n \cdot 21.885 + 366.508$$
Which means $n \geq 58$.
- Query 17: Since the time for executing this query without optimization is greater than a couple of hours, the materialization becomes useful from the first execution.

7 Mixed approach

In this section we explored a mixed approach with both materialization and indexing. As we already spared the join conditions, we will just consider indexes that are useful for slicing. Remember that the chosen materialized view results to be *lineitem-part-orders*, but for the sake of completeness, we reported also indexing on all the materialization attempts, in order to record possible time improvement.

For every materialization we will report a table with the introduced indexes and their creation time and size.

7.1 Lineitem-part

Attribute	Creation time [s]	Index size [MB]
l_shipdate	30.16	397.55
p_brand	58.95	403.14
p_container	60.34	403.15

Table 11: Indexes dimensions for *lineitem-part*

Using this indexes the total size of the materialized view is: 7.60 GB.

Remember that this materialization is useful mainly for queries 14 and 17, so the indexes were placed on the suitable slicing attributes.

7.2 Costumer-orders-lineitem-nation

Attribute	Creation time [s]	Index size [MB]
o_orderdate	58.30	397.51
l_returnflag	50.62	396.46

Table 12: Indexes dimensions for *costumer-orders-lineitem-nation*

Using this indexes the total size of the materialized view is: 13.41 GB. Recalling that this materialization is pretty useful only for one query, we exploited indexing only for the proper attributes, but given the resulting size of the table, we furthermore opted for dropping this big materialization in favor of smaller ones.

7.3 Lineitem-part-orders

Attribute	Creation time [s]	Index size [MB]
l_shipdate	47.12	397.55
p_brand	49.68	403.14
p_container	59.57	403.15

Table 13: Indexes dimensions for *lineitem-part-orders*

Using these indexes the total size of the materialized view is: 7.60 GB.

As this materialization resulted to be useful for three of the four proposed queries, we added indexes for all the useful slicing attributes involved in queries 10, 14, 17: we noticed that the indexes on *o_orderdate* and *l_returnflag* were not used, so we dropped them.

Recalling the fact that the chosen final materialization is the *lineitem-part-orders*, with all the indexes the final database size is of 22.50 GB.

We reported in table 14 the results of testing the queries with the described structure.

Query	Mean	Std
Q1	41.739	1.412
Q10	34.227	4.548
Q14	19.077	1.224
Q17	1.587	0.103

Table 14: Execution times of query with both materialized views and indexes

8 Fragmentation

We considered to implement the fragmentation only for the tables of *lineitem* and *orders* since they are the most computationally expensive to scan entirely and since they are strictly involved in the set of our chosen queries.

While designing the fragmentation, we have always considered the broader aspect of the database as a decision-making tool, avoiding introducing too specific partitions that could have improved the specific set of chosen queries, but would have unnecessarily burdened the database as they were not generalizable to other queries. Therefore, we decided to consider a temporal fragmentation, as the temporal dimension is often involved in slicing conditions, even outside of the chosen queries.

In particular we fragmented the *orders* table with respect to the *o_orderdate* attribute. Each partitioned table contains a time-span of three months to allow a significant improvement in executing query 10. Furthermore we introduced in the partitioned tables a primary key in *o_orderkey* and a foreign key on *o_custkey* referencing *c_custkey*.

Since the only keys introduced in the database with fragmentation are the one mentioned above we expect to have a database dimension which is lower than the one of the baseline.

For the *lineitem* table we decided to use a partition on *l_shipdate* and a sub-partition on *l_returnflag*, to allow exploiting this partitioning in query 1, 10, 14.

The results are reported in the table 15

Query	Mean	Std
Q1	60.109	4.050
Q10	19.369	4.091
Q14	3.834	0.160
Q17	271.702	31.483

Table 15: Execution times of query with fragmented db

9 Conclusions

The queries selected for this project were pretty different from each other, and each of them asked for a different optimization technique.

This said, we can have a comprehensive look at what happened to execution times of all the queries given all the techniques analyzed.

If we compare this behavior with the comprehensive size of the dataset, we can have a more critical opinion on what can be the best technique to adopt in each case.

Optimization	Mean	Std	DB size [GB]
Baseline	41.739	1.412	14.34
Indexes	30.009	1.224	15.32
Materialization	41.739	1.412	21.27
Mixed	41.739	1.412	22.50
Fragmentation	60.109	4.050	12.22

Table 16: Statistics of query 1

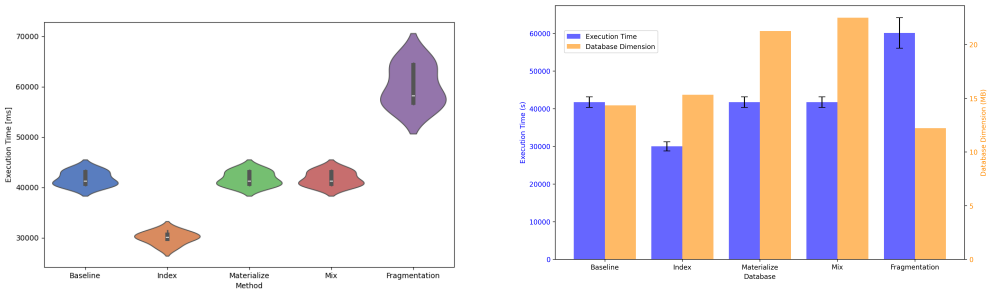

Figure 2: Query 1

Regarding query 1 we can see that the most significant improvement is reached with the usage of indexing. We can notice also that the fragmentation of the *lineitem* table worsen the performances, leading to an execution time higher than the one of the baseline. This may be expected since the slicing condition is highly non selective, and scanning (almost) all the tables in the partition have an higher cost than scanning the same quantity of rows in a single table. The performances with the strategy of materialization didn't change since the materialized views are not involved in this query.

Optimization	Mean	Std	DB size [GB]
Baseline	33.077	1.634	14.34
Indexes	25.677	1.799	15.32
Materialization	24.535	2.347	21.27
Mixed	34.227	4.548	22.50
Fragmentation	19.369	4.091	12.22

Table 17: Statistics of query 10

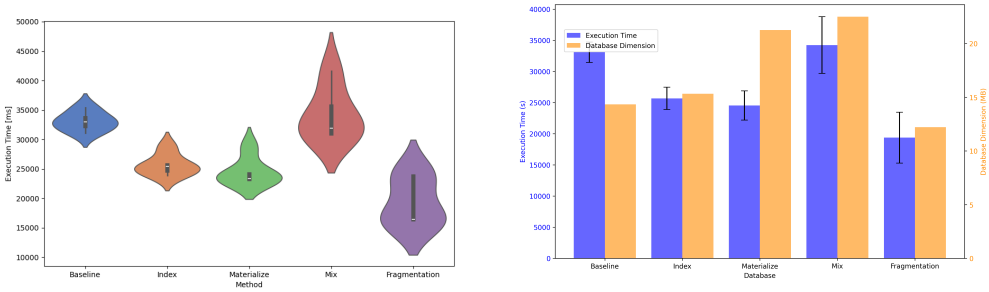

Figure 3: Query 10

Regarding query 10 we can see that the performances of all the approaches are comparable, with a slight improvement given by the fragmentation. Indeed the fragmentation on *l_orderdate* is fully exploited by the slicing condition that involved a three month linespan. Given this consideration it may be expected an even bigger improvement, but the scan of the *lineitem* partition to retrieve the subpartition with *l_returnflag="R"* moderated the gains.

Optimization	Mean	Std	DB size [GB]
Baseline	28.253	1.239	14.34
Indexes	23.607	0.327	15.32
Materialization	21.008	1.037	21.27
Mixed	19.007	1.224	22.50
Fragmentation	3.034	0.160	12.22

Table 18: Statistics of query 14

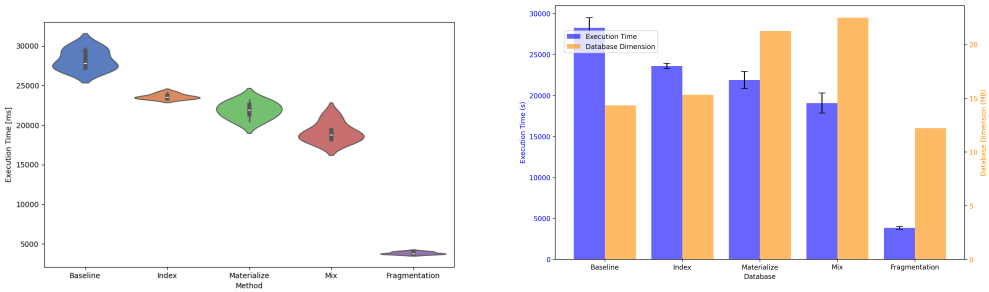

Figure 4: Query 14

Query 14 is the one that obtained the most evident improvement with the fragmentation. This is explained by the fact that the *lineitem* table is the most computationally expensive to be scanned and, by introducing the fragmentation on *l_shipdate*, the DBMS needs to consider only one sub-table, furthermore the group by condition is facilitated by the sub-partition on *l_returnflag*.

Optimization	Mean	Std	DB size [GB]
Baseline	N/A	N/A	14.34
Indexes	1.232	0.267	15.32
Materialization	20.465	0.655	21.27
Mixed	1.587	0.103	22.50
Fragmentation	271.702	31.483	12.22

Table 19: Statistics of query 17

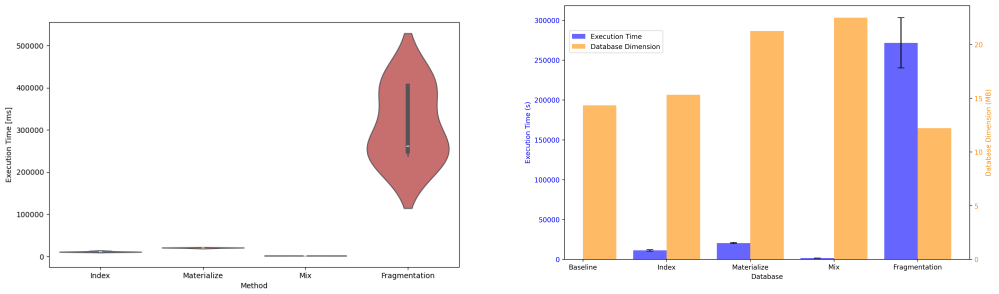

Figure 5: Query 17

Regarding query 17 all the proposed form of optimization significantly improved the execution time with respect to the baseline (initially with an execution time greater than 2 hours). A mixed approach of materialization and indexes allows to perform the slicing in a really efficient way, without having to perform a `JOIN` operation at every execution since it's included in the pre-computed view.

In conclusion, selecting a universal optimization framework for all proposed queries is complex. This task is made even more challenging by the trade-off we have to consider in terms of execution time and DB size. We can see that in absolute terms, at least with respect to time, on average the indexed materialized technique is the best in almost all the queries. Anyway, if we look at the performances that only indexes can provide, considering also their contribution to the final DB size, we can say that opting for simple indexing can seem a fairly good approach. A final word can be said if the major set of queries involves slicing into time intervals. In this context, fragmenting horizontally the DB can have a significant improvement in terms of execution time, having almost the same size of the original DB and resulting in a huge gain. Anyway, if we have some queries that involve a scan on dates that comprehend the whole table, a fragmented approach only slows down the execution.

A Appendix A

Complete statistics of the database.

Attribute	Distinct values	Min value	Max value
c_custkey	1500000	1	1500000
c_name	1500000	'Customer#000000001'	'Customer#001500000'
c_address	1500000	-	-
c_nationkey	25	0	24
c_phone	1499963	-	-
c_acctbal	818834	-999.99	9999.99
c_mktsegment	5	-	-
c_comment	1496636	-	-

Table 20: Costumer statistics

Attribute	Distinct values	Min value	Max value
s_suppkey	100000	1	100000
s_name	100000	'Supplier#000000001'	'Supplier#000100000'
s_address	100000	-	-
s_nationkey	25	0	24
s_phone	100000	-	-
s_acctbal	95588	-999.92	9999.93
s_comment	99983	-	-

Table 21: Supplier statistics

Attribute	Distinct values	Min value	Max value
p_partkey	2000000	1	2000000
p_name	1999828	-	-
p_mfgr	5	'Manufacturer#1'	'Manufacturer#5'
p_brand	25	'Brand#11'	'Brand#55'
p_type	150	-	-
p_size	50	1	50
p_container	40	-	-
p_retailprice	31681	900.91	2098.99
p_comment	806046	-	-

Table 22: Part statistics

Attribute	Distinct values	Min value	Max value
ps_partkey	2000000	1	2000000
ps_supplierkey	100000	1	100000
ps_availqty	9999	1	9999
ps_supplycost	99901	1.0	1000.0
ps_comment	7914164	-	-

Table 23: Partsupp statistics

Attribute	Distinct values	Min value	Max value
l_orderkey	15000000	1	60000000
l_partkey	2000000	1	2000000
l_supplierkey	100000	1	100000
l_linenumber	7	1	7
l_quantity	50	1	50
l_extendedprice	1351462	900.91	104949.5
l_discount	11	0.0	0.1
l_tax	9	0.0	0.08
l_returnflag	3	-	-
l_linestatus	2	-	-
l_shipdate	2526	'1992-01-02'	'1998-12-01'
l_commitdate	2466	'1992-01-31'	'1998-10-31'
l_receiptdate	2555	'1992-01-03'	'1998-12-31'
l_shipinstruction	4	-	-
l_shipmode	7	-	-
l_comment	34378943	-	-

Table 24: Lineitem statistics

Attribute	Distinct values	Min value	Max value
o_orderkey	15000000	1	60000000
o_custkey	999982	1	1499999
o_orderstatus	3	-	-
o_totalprice	11944103	838.05	558822.56
o_orderdate	2406	'1992-01-01'	'1998-08-02'
o_orderpriority	5	-	-

Table 25: Orders statistics

Attribute	Distinct values	Min value	Max value
r_regionkey	5	0	4
r_name	5	-	-
r_comment	5	-	-

Table 26: Region statistics

Attribute	Distinct values	Min value	Max value
n_nationkey	25	0	24
n_name	25	-	-
n_regionkey	25	0	4
n_comment	25	-	-

Table 27: Nation statistics