Case study:
a TPC Benchmark for decision support

Leonardo Musini

Data Management for Big Data A.Y. 2022-2023

Contents

1 Introduction

The aim of this project is to implement and improve a business oriented ad-hoc query over the public TPC-H benchmark, in order to find the best optimization strategy.

Data have been generated using dbgen package of TPC-H benchmark, that can be downloaded from TPC official website `https://www.tpc.org/tpch/`, with a scale factor of 10.
The optimization strategies that have been considered are the following:

- Indexing on tables;
- Materialization of views;
- Indexing on materialized views;
- Horizontal Fragmentation.

Each approach must keep the database size below 1.5 times the size of the original database.
The DBMS chosen for the implementation is PostgreSQL, using pgAdmin 4 management tool.

The specifications of the device used in this project are:

- Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz;
- RAM: 8.00 GB;
- Storage: SSD;
- OS: Windows 11.

2 Database definition

The database is made up of 8 tables, whose structure definitions and constraints can be found in the following documentation `https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf`. The SQL implementation is the following:

```
CREATE TABLE part (
    p_partkey INTEGER CHECK (p_partkey >= 0),
    p_name VARCHAR(55),
    p_mfgr CHAR(25),
    p_brand CHAR(10),
    p_type VARCHAR(25),
    p_size INTEGER CHECK (p_size >= 0),
    p_container CHAR(10),
    p_retailprice DECIMAL CHECK (p_retailprice >= 0),
    p_comment VARCHAR(23),
    PRIMARY KEY (p_partkey)
);

CREATE TABLE region (
    r_regionkey INTEGER CHECK (r_regionkey >= 0),
    r_name CHAR(25),
    r_comment VARCHAR(152),
    PRIMARY KEY(r_regionkey)
);

CREATE TABLE nation (
    n_nationkey INTEGER CHECK (n_nationkey >= 0),
    n_name CHAR(25),
```

```
    n_regionkey INTEGER,
    n_comment VARCHAR(152),
    PRIMARY KEY(n_nationkey),
    FOREIGN KEY(n_regionkey) REFERENCES region(r_regionkey)
);

CREATE TABLE supplier (
    s_suppkey INTEGER CHECK (s_suppkey >= 0),
    s_name CHAR(25),
    s_address VARCHAR(40),
    s_nationkey INTEGER,
    s_phone CHAR(15),
    s_acctbal DECIMAL,
    s_comment VARCHAR(101),
    PRIMARY KEY(s_suppkey),
    FOREIGN KEY (s_nationkey) REFERENCES nation(n_nationkey)
);

CREATE TABLE partsupp (
    ps_partkey INTEGER CHECK (ps_partkey >= 0),
    ps_suppkey INTEGER,
    ps_availqty INTEGER CHECK (ps_availqty >= 0),
    ps_supplycost DECIMAL CHECK (ps_supplycost >= 0),
    ps_comment VARCHAR(199),
    PRIMARY KEY (ps_partkey, ps_suppkey),
    FOREIGN KEY (ps_partkey) REFERENCES part(p_partkey),
    FOREIGN KEY (ps_suppkey) REFERENCES supplier(s_suppkey)
);

CREATE TABLE customer (
    c_custkey INTEGER CHECK (c_custkey >= 0),
    c_name VARCHAR(25),
    c_address VARCHAR(40),
    c_nationkey INTEGER,
    c_phone CHAR(15),
    c_acctbal DECIMAL,
    c_mktsegment CHAR(10),
    c_comment VARCHAR(117),
    PRIMARY KEY(c_custkey),
    FOREIGN KEY (c_nationkey) REFERENCES nation(n_nationkey)
);

CREATE TABLE orders (
    o_orderkey INTEGER,
    o_custkey INTEGER,
    o_orderstatus CHAR(1),
    o_totalprice DECIMAL CHECK (o_totalprice >= 0),
    o_orderdate DATE,
    o_orderpriority CHAR(15),
    o_clerk CHAR(15),
    o_shippriority INTEGER,
    o_comment VARCHAR(79),
    PRIMARY KEY(o_orderkey),
    FOREIGN KEY (o_custkey) REFERENCES customer(c_custkey)
);

CREATE TABLE lineitem (
    l_orderkey INTEGER,
    l_partkey INTEGER,
    l_suppkey INTEGER,
    l_linenumber INTEGER,
    l_quantity DECIMAL CHECK (l_quantity >= 0),
```

```
86      l_extendedprice DECIMAL CHECK (l_extendedprice >= 0),
87      l_discount DECIMAL CHECK (l_discount BETWEEN 0.00 AND 1.00),
88      l_tax DECIMAL CHECK (l_tax >= 0),
89      l_returnflag CHAR(1),
90      l_linestatus CHAR(1),
91      l_shipdate DATE,
92      l_commitdate DATE,
93      l_receiptdate DATE CHECK (l_shipdate <= l_receiptdate),
94      l_shipinstruct CHAR(25),
95      l_shipmode CHAR(10),
96      l_comment VARCHAR(44),
97      PRIMARY KEY (l_orderkey, l_linenumber),
98      FOREIGN KEY (l_orderkey) REFERENCES orders(o_orderkey),
99      FOREIGN KEY (l_partkey) REFERENCES part(p_partkey),
100     FOREIGN KEY (l_suppkey) REFERENCES supplier(s_suppkey),
101     FOREIGN KEY (l_partkey, l_suppkey) REFERENCES partsupp(ps_partkey, ps_suppkey)
102 );
```

2.1 Database statistics

In the following section there is a list of some statistics about the tables and some of their main attributes used in the query, except for primary keys or attributes not strictly necessary:

- `part`, with 9 columns, 2000000 rows and a size of 363 MB:
  - `p_type`, with 150 distinct values;
- `region`, with 3 columns, 5 rows and a size of 24 kB;
- `nation`, with 4 columns, 25 rows and a size of 24 kB;
- `supplier`, with 7 columns, 100000 rows and a size of 22 MB;
- `partsupp`, with 5 columns, 8000000 rows and a size of 2908 MB;
- `customer`, with 8 columns, 1500000 rows and a size of 313 MB;
- `orders`, with 9 columns, 15000000 rows and a size of 4404 MB:
  - `o_orderdate`, with 2406 distinct values, that range from 1992-01-01 to 1998-08-02;
- `lineitem`, with 16 columns, 59986052 rows and a size of 10 GB:
  - `l_extendedprice`, with 1351462 distinct values, that range from 900.91 to 104949.50,
  - `l_discount`, with 11 distinct values, that range from 0.00 to 0.10.

The total size of the database is 18 GB, then the maximum extension of the database space on the disk must not exceed 27 GB.

3 Query

The query schema asks to implement the 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 nation of the lineitem customer (export means that the supplier is in the nation **E** and import means the customer is in the nation **I**). The **revenue** is obtained by:

$$\texttt{revenue} = \texttt{l\_extendedprice} * (1 - \texttt{l\_discount})$$

The aggregations should be performed with the following roll-up :

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

The slicing is over **Type** and **Exporting nation**.

The SQL implementation is the following:

```
WITH supplier_nation_region AS (
  SELECT
    s_suppkey,
    s_name,
    s_nationkey,
    n_name AS s_nationname,
    r_name AS s_regionname
  FROM supplier
    JOIN nation ON (s_nationkey = n_nationkey)
    JOIN region ON (n_regionkey = r_regionkey)
), customer_nation_region AS (
  SELECT
    c_custkey,
    c_name,
    c_nationkey,
    n_name AS c_nationname,
    r_name AS c_regionname
  FROM customer
    JOIN nation ON (c_nationkey = n_nationkey)
    JOIN region ON (n_regionkey = r_regionkey)
), lineitem_orders AS (
  SELECT
    l_partkey,
    l_suppkey,
    o_orderdate,
    o_custkey,
    l_extendedprice * (1 - l_discount) AS revenue
  FROM lineitem JOIN orders ON (l_orderkey = o_orderkey)
)

SELECT
  EXTRACT (YEAR FROM o_orderdate) AS o_year,
  EXTRACT (QUARTER FROM o_orderdate) AS o_quarter,
  EXTRACT (MONTH FROM o_orderdate) AS o_month,
  p_type,
  s_regionname,
  s_nationname,
  s_name,
  c_regionname,
  c_nationname,
  c_name,
  SUM(revenue) AS total_revenue
FROM lineitem_orders
  JOIN part ON l_partkey = p_partkey
  JOIN supplier_nation_region ON (s_suppkey = l_suppkey)
  JOIN customer_nation_region ON (c_custkey = o_custkey)
WHERE
  s_nationkey <> c_nationkey
  AND s_nationname = 'UNITED KINGDOM'
  AND p_type = 'LARGE POLISHED STEEL'
GROUP BY
```

```
  o_year,
  o_quarter,
  o_month,
  p_type,
  c_regionname,
  c_nationname,
  c_name,
  s_regionname,
  s_nationname,
  s_name
;
```

In the previous query, since we need the nation of suppliers and the nation of customers, we create two Common Table Expressions (CTEs), `supplier_nation_region` and `customer_nation_region`, combining data respectively between `supplier` or `customer` and `nation` and `region` tables in the `JOIN` clause.
A third CTE, `lineitem_orders`, has been created as a join between `lineitem` and `order` tables in which we select the date of the orders, represented by `o_orderdate` attribute, some foreign keys needed for joining and we calculate the `revenue` as defined before.

In the main part of the query, we extract year, quarter and month from `o_orderdate`, select the attributes needed, calculate the total revenue as the sum of single group revenues, join `lineitem_orders` with `part`, `supplier_nation_region` and `customer_nation_region`, and in conjunction with the `GROUP BY` clause, we achieve the **aggregation** with the roll-up previously described.

The **slicing** is performed in the `WHERE` clause over attributes `p_type` of table `part` and `s_nationname` of `supplier_nation_region`.

4 Indexing on Tables

The first optimization was done by using indexes on tables. The `EXPLAIN ANALYZE` clause can help to understand which are the more costly operations: `JOIN`, `GROUP BY` and `WHERE` clauses are the ones that impact mostly on the query efficiency. Therefore, we have created indexes for attributes in these clauses, except for primary keys since PostgreSQL already creates indexes for them.
Here we have the implementation:

```
CREATE INDEX IF NOT EXISTS l_partkey_index
  ON lineitem (l_partkey);

CREATE INDEX IF NOT EXISTS l_suppkey_index
  ON lineitem (l_suppkey);

CREATE INDEX IF NOT EXISTS o_orderdate_index
  ON orders (o_orderdate ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS o_custkey_index
  ON orders (o_custkey);

CREATE INDEX IF NOT EXISTS p_type_index
  ON part (p_type ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS s_name_index
  ON supplier (s_name ASC NULLS LAST);
```

```
18
19 CREATE INDEX IF NOT EXISTS s_nationkey_index
20   ON supplier (s_nationkey);
21
22 CREATE INDEX IF NOT EXISTS c_name_index
23   ON customer (c_name ASC NULLS LAST);
24
25 CREATE INDEX IF NOT EXISTS c_nationkey_index
26   ON customer (c_nationkey);
27
28 CREATE INDEX IF NOT EXISTS n_regionkey_index
29   ON nation (n_regionkey);
```

The total database size is of 21 GB, respecting the size request.

5 Materialization of Views

Another way to optimize the query is to create materialized views in order to store important information into the disk and then to have a easier and faster access.
For this query we have materialized the three CTEs: in this way we can have a direct access to the tables generated from the joins in the `WITH` clause.

In the following code we show how to create the materialized views:

```
1  CREATE MATERIALIZED VIEW mv_supplier_nation_region AS
2    SELECT
3      s_suppkey,
4      s_name,
5      s_nationkey,
6      n_name AS s_nationname,
7      r_name AS s_regionname
8    FROM supplier
9      JOIN nation ON (s_nationkey = n_nationkey)
10     JOIN region ON (n_regionkey = r_regionkey);
11
12 CREATE MATERIALIZED VIEW mv_customer_nation_region AS
13   SELECT
14     c_custkey,
15     c_name,
16     c_nationkey,
17     n_name AS c_nationname,
18     r_name AS c_regionname
19   FROM customer
20     JOIN nation ON (c_nationkey = n_nationkey)
21     JOIN region ON (n_regionkey = r_regionkey);
22
23 CREATE MATERIALIZED VIEW mv_lineitem_orders AS
24   SELECT
25     l_partkey,
26     l_suppkey,
27     o_orderdate,
28     o_custkey,
29     l_extendedprice * (1 - l_discount) AS revenue
30   FROM lineitem JOIN orders ON (l_orderkey = o_orderkey);
```

We can observe some statistics of the materialized views:

- `mv_supplier_nation_region`, with 5 columns, 100000 rows and a size of 12 MB;
- `mv_customer_nation_region`, with 5 columns, 1500000 rows and a size of 157 MB;

- `mv_lineitem_orders`, with 5 columns, 59986052 rows and a size of 3390 MB.

The database has a total size of approximately 21.5 GB, so the request of 1.5 the size of the original database is respected.

Consequently, we modify the query removing the `WITH` clauses as follows:

```
SELECT
  EXTRACT (YEAR FROM o_orderdate) AS o_year,
  EXTRACT (QUARTER FROM o_orderdate) AS o_quarter,
  EXTRACT (MONTH FROM o_orderdate) AS o_month,
  p_type,
  s_regionname,
  s_nationname,
  s_name,
  c_regionname,
  c_nationname,
  c_name,
  SUM(revenue) AS total_revenue
FROM mv_lineitem_orders
  JOIN part ON l_partkey = p_partkey
  JOIN mv_supplier_nation_region ON (s_suppkey = l_suppkey)
  JOIN mv_customer_nation_region ON (c_custkey = o_custkey)
WHERE
  s_nationkey <> c_nationkey
  AND p_type = 'LARGE POLISHED STEEL'
  AND s_nationname = 'UNITED KINGDOM'
GROUP BY
  o_year,
  o_quarter,
  o_month,
  p_type,
  c_regionname,
  c_nationname,
  c_name,
  s_regionname,
  s_nationname,
  s_name
;
```

6 Indexing on Materialized Views

An improvement of the previous optimization is to create indexes on materialized views. Attributes in `JOIN`, `GROUP BY` and `WHERE` clauses were considered in this case too:

```
CREATE INDEX IF NOT EXISTS c_nationkey_index
  ON mv_customer_nation_region (c_nationkey);

CREATE INDEX IF NOT EXISTS c_regionname_index
  ON mv_customer_nation_region (c_regionname ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS c_nationname_index
  ON mv_customer_nation_region (c_nationname ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS c_name_index
  ON mv_customer_nation_region (c_name ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS s_nationkey_index
  ON mv_supplier_nation_region (s_nationkey);

```

```
CREATE INDEX IF NOT EXISTS s_regionname_index
  ON mv_supplier_nation_region (s_regionname ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS s_nationname_index
  ON mv_supplier_nation_region (s_nationname ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS s_name_index
  ON mv_supplier_nation_region (s_name ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS o_orderdate_index
  ON mv_lineitem_orders (o_orderdate ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS o_custkey_index
  ON mv_lineitem_orders (o_custkey);
```

The total size of the database is 22 GB.

7 Horizontal Fragmentation

In the `WHERE` clause of the query, we are slicing over a specific `p_type` and a specific `s_nationname`, which means we are doing a selection over those tables. Moreover, we also want that the supplier nation and customer nations are different. For this reasons, in order to reduce the execution times, we can perform an Horizontal Fragmentation (HF) over tables `part`, `supplier` and `customer`:

```
CREATE TABLE part_large_polished_steel AS (
    SELECT *
    FROM part
    WHERE p_type = 'LARGE POLISHED STEEL');

CREATE TABLE supplier_uk AS (
    SELECT supplier.*
    FROM supplier
    JOIN nation ON s_nationkey = n_nationkey
    WHERE n_name = 'UNITED KINGDOM');

CREATE TABLE customer_not_uk AS (
    SELECT customer.*
    FROM customer
    JOIN nation ON c_nationkey = n_nationkey
    WHERE n_name <> 'UNITED KINGDOM');
```

As we will see in the **Results** section, the HF shows only a slight improvement of the execution times. Then it could be useful to mix it with Vertical Fragmentation (VF), in order to reduce the workload of the query.

7.1 Hybrid and Vertical Fragmentations

Selecting only the attributes that we need in the query may help to speed up the process. We had implemented a VF on the previous tables (performing an Hybrid Fragmentation) and on the tables `lineitem` and `orders`:

```
CREATE TABLE part_large_polished_steel AS (
    SELECT p_partkey, p_type
    FROM part
    WHERE p_type = 'LARGE POLISHED STEEL');

CREATE TABLE supplier_uk AS (
    SELECT s_suppkey, s_name, s_nationkey
```

```
    FROM supplier
    JOIN nation ON s_nationkey = n_nationkey
    WHERE n_name = 'UNITED KINGDOM');

CREATE TABLE customer_not_uk AS (
    SELECT c_custkey, c_name, c_nationkey
    FROM customer
    JOIN nation ON c_nationkey = n_nationkey
    WHERE n_name <> 'UNITED KINGDOM');

CREATE TABLE lineitem_reducted AS (
    SELECT l_orderkey, l_partkey, l_suppkey, l_extendedprice, l_discount
    FROM lineitem);

CREATE TABLE orders_reducted AS (
    SELECT o_orderkey, o_custkey, o_orderdate
    FROM orders);
```

The query had to be modified in the following way:

```
WITH supplier_nation_region AS (
  SELECT
    s_suppkey,
    s_name,
    s_nationkey,
    n_name AS s_nationname,
    r_name AS s_regionname
  FROM supplier_uk
    JOIN nation ON (s_nationkey = n_nationkey)
    JOIN region ON (n_regionkey = r_regionkey)
), customer_nation_region AS (
  SELECT
    c_custkey,
    c_name,
    c_nationkey,
    n_name AS c_nationname,
    r_name AS c_regionname
  FROM customer_not_uk
    JOIN nation ON (c_nationkey = n_nationkey)
    JOIN region ON (n_regionkey = r_regionkey)
), lineitem_orders AS (
  SELECT
    l_partkey,
    l_suppkey,
    o_orderdate,
    o_custkey,
    l_extendedprice * (1 - l_discount) AS revenue
  FROM lineitem_reducted JOIN orders_reducted ON (l_orderkey = o_orderkey)
)

SELECT
  EXTRACT (YEAR FROM o_orderdate) AS o_year,
  EXTRACT (QUARTER FROM o_orderdate) AS o_quarter,
  EXTRACT (MONTH FROM o_orderdate) AS o_month,
  p_type,
  s_regionname,
  s_nationname,
  s_name,
  c_regionname,
  c_nationname,
  c_name,
  SUM(revenue) AS total_revenue
FROM lineitem_orders
```

```
  JOIN part_large_polished_steel ON l_partkey = p_partkey
  JOIN supplier_nation_region ON (s_suppkey = l_suppkey)
  JOIN customer_nation_region ON (c_custkey = o_custkey)
GROUP BY
  o_year,
  o_quarter,
  o_month,
  p_type,
  c_regionname,
  c_nationname,
  c_name,
  s_regionname,
  s_nationname,
  s_name
;
```

8 Results

In the following table the execution times of 10 runs are shown, comparing the standard query with the optimizations described previously:

Execution Times				
Standard	Indexed Tables	Materialized Views	Indexed MV	HF & VF
37.511	34.305	14.331	15.325	18.665
37.538	34.257	13.905	15.320	17.892
38.919	34.429	13.754	15.286	17.492
37.754	33.626	14.051	15.364	17.543
38.490	34.099	14.372	15.296	18.108
38.747	33.136	13.919	15.367	17.640
37.958	34.280	13.800	15.899	18.526
37.479	33.720	14.396	15.372	17.955
37.620	33.620	13.714	15.404	17.384
38.214	34.382	13.803	15.938	17.628

Table 1: Execution times are in seconds (s)

In the optimization using horizontal fragmentation on tables, we have said that just a slight improvement on the performances of the query can be observed.
In the following table we show the results of 5 runs with HF:

Horizontal Fragmentation				
36.845	37.112	36.796	37.129	36.994

Table 2: Execution times are in seconds (s)

9 Conclusions

The goal of this study is to create a database, implement the query proposed and then find the best optimization in order to achieve the best running times, without exceeding the original database size by 1.5.
We can compare the performance of the standard query with that of its optimizations in the last table, which reports means and standard deviations of execution times and sizes of the database:

	Standard	Indexed Tables	Materialized View	Indexed MV	HF & VF
μ	38.027	34.085	14.004	15.457	17.883
σ	0.511	0.410	0.254	0.233	0.415
size	18	21	21.5	22	18

Table 3: Execution times are in seconds (s), sizes are in giga-bytes (GB)

As we can observe in the table above, the best improvement in performance is reached by the query that uses materialized views, with a significant reduction in execution times. Even though we expected a further improvement using indexes on materialized views, the results shows a slight increase in executions. The fragmentation also shows very good performance improvements with the positive effect of keeping the database size constant.
In conclusion, the best optimization for this query is the one that uses materialized views that speeds the execution times approximately of 2.7 times and only increases the database size of about 1.2 times.