Data Management for Big Data

TPCH-benchmark Case Study

Simone Brusatin

September 2023

Contents

1 Introduction

The aim of the project is to implement and optimize a query which analyzes import/export revenues over the TPC-H benchmark [7] using *PostgreSQL* [5] as DBMS.

The optimization should be done using *materialized views*, *indexes* and a combination of the two; with the constraint that the size of the database after the optimization implementations should not be more than 1.5 times the original size of the database.

1.1 Software and Hardware specification

The DBMS system used is *PostgreSQL* version 15.4. *DataGrip* [1] version 2023.2.1 was used as IDE. The system runs with *Manjaro Linux* [4] operative system.

As for hardware we utilized an *HP - PC 15s-eq2012sl* laptop [3], with a AMD Ryzen™ 3 5300U (3.8 GHz max boost clock, 4 MB L3 cache, 4 cores, 8 threads) processor, 8GB of RAM and a 256GB NVMe SSD.

1.2 Methodology

Timings are repeated multiple times to reduce variance. The reported timing results is to be considered as the average, even when not specified. Different slicing values are considered in the queries also to avoid variance, although this shouldn't be a problem since the dataset is generally balanced.

2 Database

The TPC-H Benchmark [7] is a public data generator available online, that simulates orders between businesses from multiple countries of different items.

A detailed description of the schema is provided in the website [6]. The data is generated according to a *scale factor* that regulates the size of the tables. In our particular case, the scale factor is equal to 10.

The relation schema is provided in figure 1.

2.1 SQL Implementation

The relational schema has been converted in *SQL's Data Definition Language* using the code below 1.

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

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

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
);

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
);

CREATE TABLE PARTSUPP(
```

```sql
    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
);

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
);

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
);

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
);
```

Listing 1: SQL implementation of tables

Then the primary and foreign keys are added with the following code 2.

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

ALTER TABLE SUPPLIER
    ADD PRIMARY KEY (S_SUPPKEY);

ALTER TABLE PARTSUPP
    ADD PRIMARY KEY (PS_PARTKEY, PS_SUPPKEY);

```

```sql
ALTER TABLE CUSTOMER
    ADD PRIMARY KEY (C_CUSTKEY);

ALTER TABLE ORDERS
    ADD PRIMARY KEY (O_ORDERKEY);

ALTER TABLE LINEITEM
    ADD PRIMARY KEY (L_ORDERKEY, L_LINENUMBER);

ALTER TABLE NATION
    ADD PRIMARY KEY (N_NATIONKEY);

ALTER TABLE REGION
    ADD PRIMARY KEY (R_REGIONKEY);

ALTER TABLE supplier
    ADD FOREIGN KEY (s_nationkey) REFERENCES nation(N_NATIONKEY);

ALTER TABLE partsupp
    ADD FOREIGN KEY (ps_partkey) REFERENCES part(p_partkey);

ALTER TABLE partsupp
    ADD FOREIGN KEY (ps_suppkey) REFERENCES supplier(s_suppkey);

ALTER TABLE customer
    ADD FOREIGN KEY (c_nationkey) REFERENCES nation(n_nationkey);

ALTER TABLE orders
    ADD FOREIGN KEY (o_custkey) REFERENCES  customer(c_custkey);

ALTER TABLE lineitem
    ADD FOREIGN KEY (l_orderkey) REFERENCES orders(o_orderkey);

ALTER TABLE lineitem
    ADD FOREIGN KEY (l_partkey) REFERENCES part(p_partkey);

ALTER TABLE lineitem
    ADD FOREIGN KEY (l_suppkey) REFERENCES supplier(s_suppkey);

ALTER TABLE lineitem
    ADD FOREIGN KEY (l_partkey, l_suppkey) REFERENCES partsupp(ps_partkey, ps_suppkey);

ALTER TABLE nation
    ADD FOREIGN KEY (n_regionkey) REFERENCES region(r_regionkey);
```

Listing 2: SQL implementation of keys

The tables have been subsequently populated with the generated data through the IDE's in-built command.

2.2 Database Statistics

In the following table 2.2 we report the size of the the tables and of the whole database, both in terms of rows and in bytes. We also report the size of the primary indexes. Note that ulterior indexes are not yet implemented.

	Rows	Total Size	Table Size	Index Size
CUSTOMER	1,500,000	312 MB	280 MB	32 MB
LINEITEM	59,986,052	10073 MB	8788 MB	1285 Mb
NATION	25	24 kB	8192 bytes	16 kB
ORDERS	15,000,000	2360 MB	2039 MB	321 MB
PART	2,000,000	363 MB	320 MB	43 MB
PARTSUPP	8,000,000	1534 MB	1363 MB	171 MB
REGION	5	24 kB	8192 bytes	16 kB
SUPPLIER	100,000	20 MB	17 MB	2208 kB
TOTAL	86,586,082	14662 MB	12807 MB	1855 MB

In the next table 2.2, we report the count of distinct values for each, non-primary, attribute used for querying. Moreover we also report minimum and maximum value for numerical attributes. Note that the table containing the attribute is identified by the prefix in the attribute name.

	distinct values	min value	max value
l_extendedprice	1351462	900.91	104949.5
l_discount	11	0	0.1
n_name	25		
o_orderdate	2406	1992-01-01	1998-08-02
p_type	150		
r_name	5		

3 Query

In this section we propose a solution for the first query requested in the assignment: *export/import revenue value*. The request is as follows:

> 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.

3.1 SQL query

The query is implemented in *SQL* with the following code 3

```sql
CREATE VIEW exp AS
SELECT
    l_orderkey AS e_orderkey,
    l_partkey AS e_partkey,
    l_suppkey AS e_suppkey,
```

```sql
    l_extendedprice AS e_extendedprice,
    l_discount AS e_discount,
    p_type AS e_type,
    n_nationkey AS e_nationkey,
    n_name AS e_nname,
    r_regionkey AS e_regionkey,
    r_name AS e_rname
FROM lineitem
    JOIN part ON lineitem.l_partkey = part.p_partkey
    JOIN supplier ON lineitem.l_suppkey = supplier.s_suppkey
    JOIN nation ON supplier.s_nationkey = nation.n_nationkey
    JOIN region ON nation.n_regionkey = region.r_regionkey;


CREATE VIEW imp AS
SELECT
    l_orderkey AS i_orderkey,
    l_partkey AS i_partkey,
    l_suppkey AS i_suppkey,
    l_extendedprice AS i_extendedprice,
    l_discount AS i_discount,
    o_orderdate AS i_date,
    n_nationkey AS i_nationkey,
    n_name AS i_nname,
    r_regionkey AS i_regionkey,
    r_name AS i_rname
FROM lineitem
    JOIN orders ON lineitem.l_orderkey = orders.o_orderkey
    JOIN customer ON orders.o_custkey = customer.c_custkey
    JOIN nation ON customer.c_nationkey = nation.n_nationkey
    JOIN region ON nation.n_regionkey = region.r_regionkey;


--EXPLAIN ANALYSE
SELECT
    DATE_PART('month', i_date) AS month,
    DATE_PART('quarter', i_date) AS quarter,
    DATE_PART('year', i_date) AS year,
    e_type AS type,
    e_nname AS export_nation,
    e_rname AS export_region,
    i_nname AS import_nation,
    i_rname AS import_region,
    SUM(e_extendedprice * (1 - e_discount)) AS revenue
FROM exp
    JOIN imp ON e_orderkey = i_orderkey AND e_partkey = i_partkey AND e_suppkey = i_suppkey
WHERE e_type = 'ECONOMY ANODIZED BRASS' AND e_nname = 'ETHIOPIA'
GROUP BY ROLLUP(year, quarter, month),
         e_type,
         ROLLUP(export_region, export_nation),
         ROLLUP(import_region, import_nation);
```

Listing 3: SQL implementation of the qury

In figure 2, we report an example of a partial result of the query execution. The execution of the query required on average $40.9s \pm 0.8s$.

4 Optimization

In this section we apply different optimization methods: indexes, materialized views and a combination of the two. We will report timings. An all out comparison will be performed in the next section.

4.1 Indexes

To understand which indexes can be useful, we created indexes on all the variables utilized in the query. Then, with the `EXPLAIN ANALYSE` command, we analysed which indexes were actually used.

In the listing below 4 we can see the code utilized to create all indexes[1]. The only utilized indexes are those on `l_partkey`, `p_type` and `s_nation`. Unused indexes where discarded to reduce planning time.

```sql
-- indexes on tables

CREATE INDEX l_partkey_index ON lineitem(l_partkey); --used
CREATE INDEX l_suppkey_index ON lineitem(l_suppkey);
CREATE INDEX l_orderkey_index ON lineitem(l_orderkey);

CREATE INDEX o_orderkey_index ON orders(o_orderkey);
CREATE INDEX o_custkey_index ON orders(o_custkey);

CREATE INDEX c_custkey_index ON customer(c_custkey);
CREATE INDEX c_nationkey_index ON customer(c_nationkey);

CREATE INDEX p_partkey_index ON part(p_partkey);
CREATE INDEX p_type_index ON part(p_type); --used

CREATE INDEX s_suppkey_index ON supplier(s_suppkey);
CREATE INDEX s_nationkey_index ON supplier(s_nationkey); --used

CREATE INDEX n_nationkey_index ON nation(n_nationkey);
CREATE INDEX n_regionkey_index ON nation(n_regionkey);
CREATE INDEX n_name_index ON nation(n_name);

CREATE INDEX r_regionkey_index ON region(r_regionkey);

-- indexes on materialized views

CREATE INDEX e_orderkey_ind ON exp(e_orderkey);
CREATE INDEX e_partkey_ind ON exp(e_partkey);
CREATE INDEX e_suppkey_ind ON exp(e_suppkey);
CREATE INDEX e_type_ind ON exp(e_type); --used
CREATE INDEX e_nname_ind ON exp(e_nname); --used
```

[1]indexes on materialized views are also present but can be ignored until section 4.3

```sql

CREATE INDEX i_orderkey_ind ON imp(i_orderkey); --used
CREATE INDEX i_partkey_ind ON imp(i_partkey); --used
CREATE INDEX i_suppkey_ind ON imp(i_suppkey);
CREATE INDEX i_nnami_ind ON imp(i_nname);
```

Listing 4: SQL implementation of indexes

In the first table 4.1 we report the space occupied by the indexes, compared with the original size occupied by the related tables. The space used by the index satisfies the constraint.

	Index Size	New Total Size	Old Total Size
l_partkey	430 MB	10503 MB	10073 MB
p_type	14 MB	377 MB	363 MB
s_nationkey	704 kB	20 MB	20 MB

In the second table 4.1 we compare the execution time of the query with and without indexes. As we can see, there is no improvement. The indexes actually worsen performances by 14%.

Ex. Time with Indexes	Ex. Time without Indexes	Improvement
46840.17 ms	40911.84 ms	-5928.33 ms

4.2 Materialized Views

Since the original query (3) already utilizes virtual views, it is sufficient to add the keyword `MATERIALIZED` to obtain the materialized views.

In the table 4.2 we can see the space consumption of the two views. As we can see, the materialization requires about ≈ 1.15 times the original space of the database, well below the 1.5 constraint.

	Size
exp	8224 MB
imp	7212 MB
mat views	15 GB
relations	13 GB

Regarding the execution time, we can see in table 4.2 a substantial improvement, requiring less than a third of the original time.

Ex. Time with Mat. Views	Ex. Time w/o Mat. Views	Improvement
12003.53 ms	40911.84 ms	28908.31 ms

4.3 Indexes on Materialized Views

Finally we are going to study the case where we place indexes on the materialized views created in the previous section (4.2).

We create indexes with the same procedure as seen previously in listing 4. The indexes utilized by the optimizer are `e_type` and `e_nname` on view `exp`, `i_orderkey` and `i_partkey` on view `imp`. All other indexes are discarded to save space.

In the table 4.3 below we report the size of the indexes. Moreover, we report the size of the materialized view comprehensive of the indexes compared with the size without. The total size of the materialization is now 17426MB. Compared with the original size of 14662MB, it occupies about 19% more space.

	Index Size	View Size (with index)	View Size (original)
e_type	407 MB		
e_nname	411 MB	9042 MB	8224 MB
i_orderkey	741 MB		
i_partkey	430 MB	8384 MB	7212 MB

In the next table 4.3, we can see instead the average time of the query execution. It shows a large improvement over the execution time of the query without optimization. It also shows a small improvement over the materialized version without indexes.

Ex. Time with Optimization	Ex. Time w/o Optimization	Improvement
10442.32 ms	40911.84 ms	30469.52 ms

5 Conclusions

In this section we compare all the optimization strategies and discuss which one is the most suitable. In the table 5 below we can see the time comparisons of all techniques.

	Ex. Time	JIT Time	Planning Time	Total Time
No Optimization	40911.8 ± 767.4 ms	2586.5 ± 249.6 ms	1.5 ± 0.4 ms	43499.9 ± 953.9 ms
Indexes	46840.1 ± 641.9 ms	2200.9 ± 40.5 ms	7.2 ± 0.7 ms	49048.4 ± 649.7 ms
Materialized Views	12003. ± 736. ms	1842.0 ± 161.8 ms	0.4 ± 0.2 ms	13846.1 ± 893.4 ms
Indexed Mat. Views	10442.3 ± 35.5 ms	91.9 ± 4.8 ms	5.8 ± 0.7 ms	10540.0 ± 36.5 ms

The best optimization approach is to create materialized views and build indexes over them. This not only decreases execution time considerably but also reduces *JIT* [2] time to almost zero, thus granting an even lower total time. It should also be noted that the variance is also substantially decreased with this approach.

It has to be considered though that the implementation of the indexed materialized views is the most expensive in terms of disk space. To fully assess which optimization strategy is the most suitable, we should also consider how frequently the query is intended to be executed and which other queries are requested from the database.

[2] Just-in-Time (JIT) compilation is the process of turning some form of interpreted program evaluation into a native program, and doing so at run time. For example, instead of using general-purpose code that can evaluate arbitrary SQL expressions to evaluate a particular SQL predicate like `WHERE a.col = 3`, it is possible to generate a function that is specific to that expression and can be natively executed by the CPU, yielding a speedup. [2]

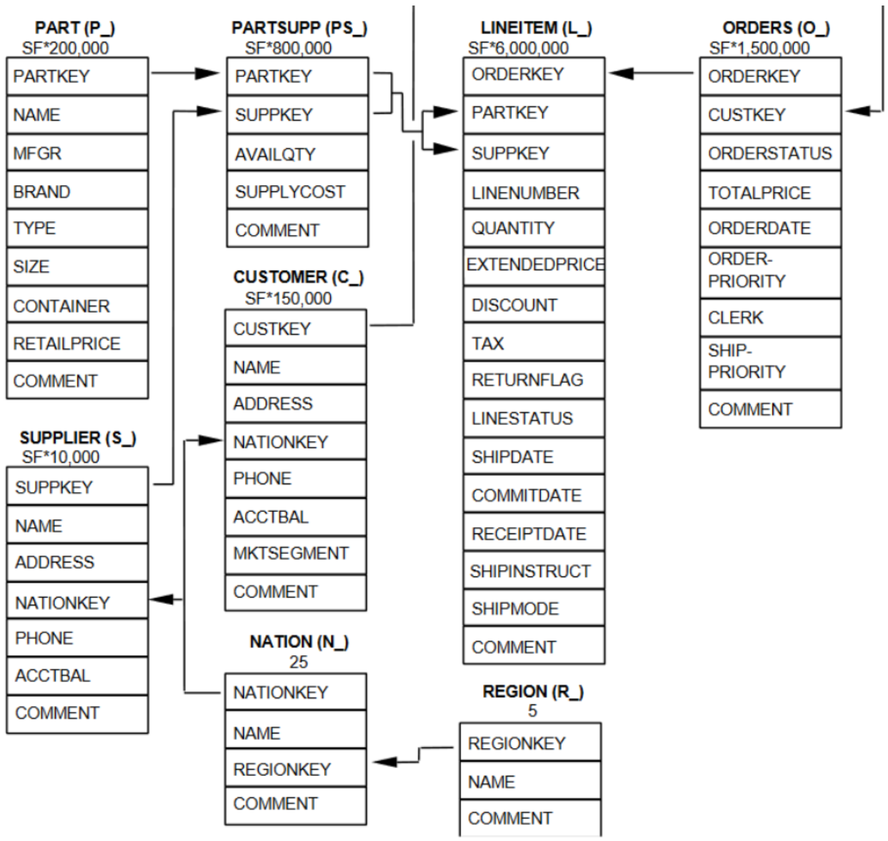

Figure 1: Relational Schema

	month	quarter	year	type	export_nation	export_region	import_nation	import_region	revenue
1	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	VIETNAM	ASIA	3003.7632
2	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	BRAZIL	AMERICA	9305.8848
3	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	INDONESIA	ASIA	10612.0800
4	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA		ASIA	13615.8432
5	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	KENYA	AFRICA	13679.1468
6	11	4	1996	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ETHIOPIA	AFRICA	16008.1812
7	12	4	1992	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	BRAZIL	AMERICA	17998.3440
8	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	FRANCE	EUROPE	20721.7692
9	9	3	1997	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ETHIOPIA	AFRICA	25633.1521
10	1	1	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ROMANIA	EUROPE	30377.3528
11	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	GERMANY	EUROPE	31427.1173
12	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ARGENTINA	AMERICA	32916.0176
13	2	1	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ARGENTINA	AMERICA	36040.7080
14	4	2	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	RUSSIA	EUROPE	39349.9200
15	8	3	1992	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	JAPAN	ASIA	40152.2130
16	8	3	1998	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ALGERIA	AFRICA	42524.1180
17	11	4	1993	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	JAPAN	ASIA	43284.2867
18	11	4	1995	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	CANADA	AMERICA	43542.2655
19	9	3	1996	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ROMANIA	EUROPE	46279.7316
20	12	4	1992	ECONOMY ANODIZED BRASS	ETHIOPIA	AFRICA	ROMANIA	EUROPE	46783.1034

Figure 2: Output Example

Bibliography

[1] *DataGrip Website*. URL: https://www.jetbrains.com/datagrip/.

[2] *JIT Documentation*. URL: https://www.postgresql.org/docs/current/jit-reason.html#JIT-REASON.

[3] *Laptop Specification*. URL: https://support.hp.com/sg-en/document/c08484249.

[4] *Manjaro Website*. URL: https://manjaro.org/.

[5] *PostgreSQL Website*. URL: https://www.postgresql.org/.

[6] *TPC Description*. URL: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf.

[7] *TPC Website*. URL: https://www.tpc.org/tpch/.