University of Trieste

Data Management for Big Data Course

Academic Year 2022–2023

Data Warehouse case study

Davide Capone* Sandro Junior Della Rovere† Enrico Stefanel‡

Last update on 2023–06–25T13:15:58Z.

Contents

*davide.capone@studenti.units.it
†sandrojunior.dellarovere@studenti.units.it
‡enrico.stefanel@studenti.units.it

1 Introduction

The aim of this project is to study an efficient implementation of a suite of business oriented ad-hoc queries over the public TPC-H benchmark, which can be considered as a Big Data database, that has been implemented in Postgres.

1.1 TPC-H benchmark database

The TPC-H benchmark is a decision support benchmark that can be downloaded from the TPC official website. The data generator lets the user specify a *scale factor* in order to control the size of the resulted database. Our choices was to use a *scale factor* of 10, meaning that the overall database size is approximately 13 GB.

1.1.1 Database statistics

The benchmark is composed by eight tables:

- `CUSTOMER`, with 16 columns and 1 500 000 tuples (312 MB);
- `LINEITEM`, with 32 columns and 59 986 052 tuples (11 GB); the main attributes that are going to be used are:
  - `l_extendedprice` (1 351 462 distinct values, i.e. there is an average of 44 tuples with the same value, that range from 900.91 to 104 949.50),
  - `l_discount` (11 distinct values, i.e. there is an average of 5 453 277 tuples with the same value, that range from 0.00 to 0.10),
  - `l_returnflag` (which can assume values `A`→accepted, `R`→returned, `N`→not yet delivered; the percentage of tuples for `A` and `R` are almost 25 %, while the percentage of tuples where `l_returnflag` is `N` is about 50 %),
  - `l_commitdate` (2466 distinct values, i.e. there is an average of 24 325 tuples with the same value, that range from 1992-01-31 to 1998-10-31),
  - `l_receiptdate` (2555 distinct values, i.e. there is an average of 23 478 tuples with the same value, that range from 1992-01-03 to 1998-12-31);
- `NATION`, with 8 columns and 25 tuples (24 kB);

- `ORDERS`, with 18 columns and 1 500 000 tuples (2481 kB); the main attributes that are going to be used are:
  - `o_orderdate` (2406 distinct values, i.e. there is an average of 6234 tuples with the same value, that range from 1992-01-01 to 1998-08-02);
- `PART`, with 18 columns and 2 000 000 tuples (363 MB); the main attributes that are going to be used are:
  - `p_type` (150 distinct values, i.e. there is an average of 13 333 tuples with the same value);
- `PARTSUPP`, 10 columns and with 8 000 000 tuples (1535 MB);
- `REGION`, 6 columns and with 5 tuples (24 kB);
- `SUPPLIER`, 14 columns and with 100 000 tuples (20 MB).

Other attributes have been used, but statistics about them have been omitted for lack of usefulness (e.g., keys of the tables, for which the cardinality is exactly the cardinality of the corresponding table).

1.1.2 Database SQL definition

The SQL definition of the tables can be found on the official benchmark download.

1.2 Organization of work

Each of the group component implemented a query (specifically, Della Rovere designed the first query, Capone on the second one, and Stefanel the last one). After regular meetings between the group members comparing the solo works, common substructures between queries have been detected, and the following work proceed in a coral way.

All the execution times reported in the present report are the results of runs on the same machine, with roughly the same external factors. In particular, a computer with an Apple M1 processor, 8 GB of RAM, and macOS operating system has been used.

2 Set of queries

2.1 Export/import revenue value

It is asked to return the *export/import revenue* between two different nations (E, I) where E is the nation of the lineitem supplier and I the nation of the lineitem customer, and where the *revenue* is defined as

```
SUM(l_extendedprice*(1-l_discount))
```

. The aggregation should be performed with the Month → Quarter → Year, (Part) Type and Nation → Region roll-ups.

The query is implemented in such a way that it allows the slicing over (Part) Type and Exporting Nation.

```
WITH lineitem_orders AS (
    SELECT
        l_partkey,
        l_suppkey,
        o_orderdate,
        o_custkey,
        l_extendedprice,
        l_discount
    FROM lineitem JOIN orders ON (l_orderkey = o_orderkey)
), customer_location AS (
    SELECT
        c_custkey,
        c_name,
        n_nationkey AS c_nationkey,
        n_name AS c_nationname,
        r_regionkey AS c_regionkey,
        r_name AS c_regionname
    FROM customer
        JOIN nation ON (c_nationkey = n_nationkey)
        JOIN region ON (n_regionkey = r_regionkey)
), supplier_location AS (
    SELECT
        s_suppkey,
        s_name,
        n_nationkey AS s_nationkey,
        n_name AS s_nationname,
        r_regionkey AS s_regionkey,
        r_name AS s_regionname
    FROM supplier
        JOIN nation ON (s_nationkey = n_nationkey)
        JOIN region ON (n_regionkey = r_regionkey)
), query1 AS (
    SELECT
        EXTRACT (YEAR FROM o_orderdate) AS _year,
        EXTRACT (QUARTER FROM o_orderdate) AS _quarter,
        EXTRACT (MONTH FROM o_orderdate) AS _month,
        c_regionname,
        c_nationname,
        c_name,
```

```
        s_regionname,
        s_nationname,
        s_name,
        p_type,
        SUM(l_extendedprice * (1 - l_discount)) AS revenue
    FROM lineitem_orders
        JOIN part ON l_partkey = p_partkey
        JOIN supplier_location ON (s_suppkey = l_suppkey)
        JOIN customer_location ON (c_custkey = o_custkey)
    WHERE
        s_nationkey <> c_nationkey
        -- AND p_type = 'PROMO BURNISHED COPPER'
        -- AND s_nationname = 'UNITED STATES'
    GROUP BY
        _year,
        _quarter,
        _month,
        c_regionkey,
        c_regionname,
        c_nationkey,
        c_nationname,
        c_custkey,
        c_name,
        s_regionkey,
        s_regionname,
        s_nationkey,
        s_nationname,
        s_suppkey,
        s_name,
        p_type
)
SELECT * FROM query1;
```

2.2 Late delivery

It is asked to retrieve the number of orders where at least one “lineitem” has been received later than the committed date.

The aggregation should be performed with the Month → Year roll-up, and the (Customer's) Nation → Region roll-up.

The query has been implemented in such a way that it allows the slicing over a specific Month, and a specific (Part) Type.

```
WITH lineitem_orders AS (
```

```sql
    SELECT
        o_orderkey,
        l_partkey,
        l_suppkey,
        o_orderdate,
        o_custkey,
        l_commitdate,
        l_receiptdate
    FROM lineitem JOIN orders ON (l_orderkey = o_orderkey)
), customer_location AS (
    SELECT
        c_custkey,
        n_nationkey AS c_nationkey,
        n_name AS c_nationname,
        r_regionkey AS c_regionkey,
        r_name AS c_regionname
    FROM customer
        JOIN nation ON (c_nationkey = n_nationkey)
        JOIN region ON (n_regionkey = r_regionkey)
), query2 AS (
SELECT
    EXTRACT(YEAR FROM o_orderdate) AS _year,
    EXTRACT(MONTH FROM o_orderdate) AS _month,
    c_regionname,
    c_nationname,
    COUNT(DISTINCT(o_orderkey)) AS orders_no
FROM lineitem_orders
    JOIN part ON l_partkey = p_partkey
    JOIN customer_location ON (c_custkey = o_custkey)
WHERE
    l_receiptdate > l_commitdate
    -- AND _month = 1
    -- AND p_type = 'PROMO BURNISHED COPPER'
GROUP BY
    _year,
    _month,
    c_regionkey,
    c_regionname,
    c_nationkey,
    c_nationname
)
SELECT * FROM query2;
```

2.3 Returned item loss

It is asked to retrieve the *revenue loss* for customers who might be having problems with the parts that are shipped to them, where a *revenue loss* is defined as

`SUM(l_extendedprice*(1-l_discount))`

for all qualifying *lineitems*.

The aggregations should be performed with the Month → Quarter → Year and Customer roll-ups.

The query has been implemented in such a way that it allows the slicing over the Name of a Customer combined with a specific Quarter.

```sql
WITH lineitem_orders AS (
    SELECT
        o_orderkey,
        o_orderdate,
        o_custkey,
        l_extendedprice,
        l_discount,
        l_returnflag
    FROM lineitem JOIN orders ON (l_orderkey=o_orderkey)
),
query3 AS (
SELECT
    EXTRACT(YEAR FROM o_orderdate) AS _year,
    EXTRACT(QUARTER FROM o_orderdate) AS _quarter,
    EXTRACT(MONTH FROM o_orderdate) AS _month,
    c_name,
    SUM(l_extendedprice*(1-l_discount)) AS returnloss
FROM
    lineitem_orders
    JOIN customer ON (o_custkey=c_custkey)
WHERE
    l_returnflag='R'
    -- AND c_name='Customer#000129976'
    -- AND EXTRACT(QUARTER FROM o_orderdate) = 1
GROUP BY
    _year,
    _quarter,
    _month,
    c_custkey,
    c_name
)
SELECT * FROM query3;
```

2.4 Execution times

The query timings have been measured as previously discussed in subsection 1.2. Furthermore, no consecutive runs have been performed on the same query, in order to reduce external biases (following a Round-Robin schema).

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	40 944	39 423	41 053	39 928	38 800	40 029	971
2	45 150	51 300	46 270	46 677	50 235	47 626	2679
3	8245	6886	8604	8790	6943	7893	915

Table 1: Naïve query timings, in milliseconds.

3 Materialization

After a study on the given set of queries, some common intermediate results have been detected between the three. In order to try lowering the average execution cost, materialized views have been defined starting from the said results.

Furthermore, since the problem considers dealing with a data warehouse (OLAP), there won't be frequent updates, so materialized views are a smart choice.

3.1 Lineitem - Orders View

Since all the queries perform a join between relations *lineitem* and *orders*, an idea is to pre-process this join by creating a materialized view.

```
CREATE MATERIALIZED VIEW lineitem_orders_mv AS
    SELECT
        o_orderkey,
        l_partkey,
        l_suppkey,
        o_orderdate,
        o_custkey,
        l_extendedprice,
        l_discount,
        l_returnflag,
        l_commitdate,
        l_receiptdate
    FROM lineitem JOIN orders ON (l_orderkey = o_orderkey);
```

This materialized view is composed by 59 986 052 rows, and weighs 4378 MB.

3.1.1 Customer - Location View

Queries 1 and 2 execute a join operation between *customer*, *nation* and *region*, also this step has been pre-processed by creating a materialized view.

```sql
CREATE MATERIALIZED VIEW customer_location_mv AS
    SELECT
        c_custkey,
        c_name,
        n_nationkey AS c_nationkey,
        n_name AS c_nationname,
        r_regionkey AS c_regionkey,
        r_name AS c_regionname
    FROM customer
        JOIN nation ON (c_nationkey = n_nationkey)
        JOIN region ON (n_regionkey = r_regionkey);
```

This materialized view is composed by 1 500 000 rows, and weighs 167 MB.

3.1.2 Supplier - Location View

Finally, queries 1 and 2 execute a join operation between *supplier*, *nation* and *region*, also this step has been pre-processed by creating a materialized view.

```sql
-- used by Q1 and Q2
CREATE MATERIALIZED VIEW supplier_location_mv AS
    SELECT
        s_suppkey,
        s_name,
        n_nationkey AS s_nationkey,
        n_name AS s_nationname,
        r_regionkey AS s_regionkey,
        r_name AS s_regionname
    FROM supplier
        JOIN nation ON (s_nationkey = n_nationkey)
        JOIN region ON (n_regionkey = r_regionkey);
```

This materialized view is composed by 100 000 rows, and weighs 12 MB.

3.2 Execution times

As for the base versions of queries, we collected statistics on execution timings and results can be observed on Table 4. Further considerations are reported in the section 5.

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	16 207	15 048	15 257	15 421	15 028	15 392	483
2	14 957	17 235	15 519	16 654	16 400	16 153	910
3	13 742	15 604	14 050	14 822	14 822	14 608	732

Table 2: Query timings with materialized views, in milliseconds.

4 Indexes design

Indexes may help to further reduce the query execution times. In order to design the indexes, the queries have been executed with the `EXPLAIN ANALYSE` tool, which helps to detect the most expensive operations that are involved.

It turns out that the `JOIN` and `GROUP BY` operations are the most costly, so the indexes were built on the attributes involved in the aforementioned operations and also on the ones involved in `WHERE` clauses. Only attributes that are used by at least two queries have been considered.

4.1 Indexes on relations

```sql
CREATE INDEX IF NOT EXISTS lineitem_l_orderkey_idx
    ON lineitem USING btree
    (l_orderkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS lineitem_l_suppkey_idx
    ON lineitem USING btree
    (l_suppkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS lineitem_l_partkey_idx
    ON lineitem USING btree
    (l_partkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS order_o_orderdate_idx
    ON orders USING btree
    (o_orderdate ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS order_o_custkey_idx
    ON orders USING btree
    (o_custkey ASC NULLS LAST)
    TABLESPACE pg_default;
```

```

CREATE INDEX IF NOT EXISTS part_p_type_idx
    ON part USING btree
    (p_type ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS nation_n_name_idx
    ON nation USING btree
    (n_name ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS region_r_name_idx
    ON region USING btree
    (r_name ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS supplier_s_nationkey_idx
    ON supplier USING btree
    (s_nationkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_c_nationkey_idx
    ON customer USING btree
    (c_nationkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_c_name_idx
    ON customer USING btree
    (c_name ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.1.1 Execution times

The tests have been performed without the materialized views. In subsection 4.2 also materialized views are going to be considered.

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	643	685	724	698	813	713	63
2	201	203	205	203	203	203	2
3	98	102	101	102	101	101	2

Table 3: Query timings with indexes, in seconds.

4.2 Indexes on Materialized Views

Trying to additionally cut the query costs, also indexes on the materialized views have been designed, following the same strategy as before.

```
CREATE INDEX IF NOT EXISTS lineitem_orders_o_orderkey_idx
    ON lineitem_orders_mv USING btree
    (o_orderkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS lineitem_orders_l_suppkey_idx
    ON lineitem_orders_mv USING btree
    (l_suppkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS lineitem_orders_l_partkey_idx
    ON lineitem_orders_mv USING btree
    (l_partkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS lineitem_orders_o_orderdate_idx
    ON lineitem_orders_mv USING btree
    (o_orderdate ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS lineitem_orders_o_custkey_idx
    ON lineitem_orders_mv USING btree
    (o_custkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS supplier_location_s_nationkey_idx
    ON supplier_location_mv USING btree
    (s_nationkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS supplier_location_s_nationname_idx
    ON supplier_location_mv USING btree
    (s_nationname ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS supplier_location_s_regionkey_idx
    ON supplier_location_mv USING btree
    (s_regionkey ASC NULLS LAST)
    TABLESPACE pg_default;
```

```

CREATE INDEX IF NOT EXISTS supplier_location_s_regionname_idx
    ON supplier_location_mv USING btree
    (s_regionname ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_location_c_nationkey_idx
    ON customer_location_mv USING btree
    (c_nationkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_location_c_nationname_idx
    ON customer_location_mv USING btree
    (c_nationname ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_location_c_regionkey_idx
    ON customer_location_mv USING btree
    (c_regionkey ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_location_c_regionname_idx
    ON customer_location_mv USING btree
    (c_regionname ASC NULLS LAST)
    TABLESPACE pg_default;

CREATE INDEX IF NOT EXISTS customer_location_c_name_idx
    ON customer_location_mv USING btree
    (c_name ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.2.1 Execution times

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	643	685	724	698	813	713	63
2	201	203	205	203	203	203	2
3	98	102	101	102	101	101	2

Table 4: Query timings with indexes and materialized views, in seconds.

5 Conclusions

Siccome nelle viste materializzate ci sono più attributi rispetto alle tabelle intermedie con WITH, nella prima query, allora quella query è più pesante con le viste materializzate.

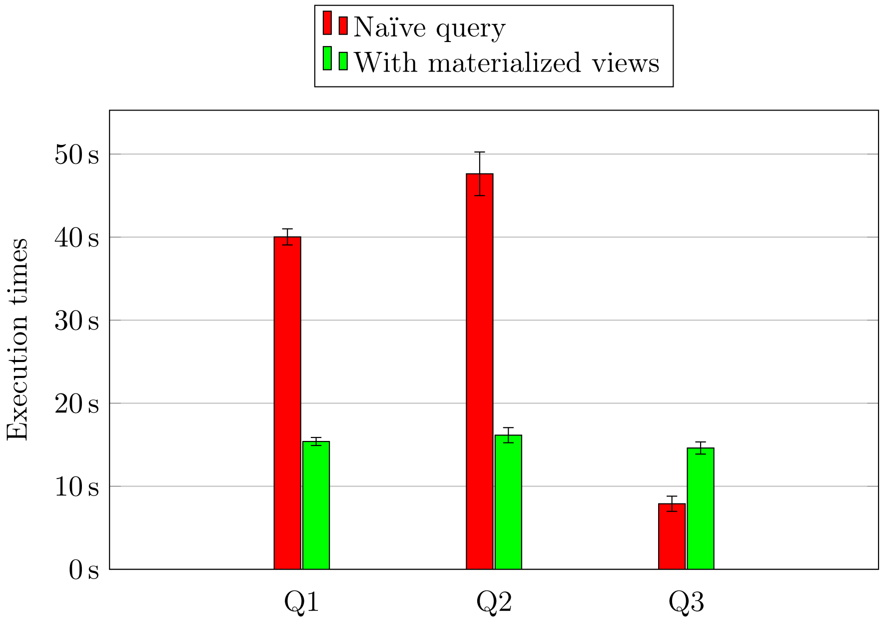

Figure 1: Query timings