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

Contents

Last update on 2023–06–28T11:23:11Z.

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 datawarehouse. The chosen DMBS for the implementation is PostgreSQL.

The way in which this efficient implementation is going to be realized is by exploiting *materialized views*, *indexes* and *(vertical) fragmentation*. The total weight of the database after the implementation of these solutions is requested to stay below 1.5 times the original database weight.

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 resulting database. Our choices was to use a *scale factor* of 10, meaning that the overall database size is approximately 14 GB. The sizes of the individual tables that compose the database are also shown below.

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 attribute that is going to be used is:
  - `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 attribute that is going to be used is:
  - 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 documentation[1].

1.2 Organization of work

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

All the execution times reported in the present report are the results of independent 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. Furthermore, no consecutive runs have been performed on the same query, in order to reduce external biases (following a Round-Robin schema).

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.

```sql
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 (
    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.

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, and they have been chosen as candidate views. 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 would not 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 primary 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.

3.2 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 primary view.

```
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.

3.3 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 another primary view.

```
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.

3.4 Queries with materialized views

The original queries have been re-written, in order to use materialized views instead of using `JOIN` operations on the related tables.

```
WITH 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_mv
        JOIN part ON l_partkey = p_partkey
        JOIN supplier_location_mv ON (s_suppkey = l_suppkey)
        JOIN customer_location_mv 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;
```

```
WITH 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_mv
    JOIN part ON l_partkey = p_partkey
    JOIN customer_location_mv ON (c_custkey = o_custkey)
WHERE
    l_receiptdate > l_commitdate
    AND EXTRACT(MONTH FROM o_orderdate) = 1
    AND p_type = 'PROMO BURNISHED COPPER'
GROUP BY
    _year,
    _month,
    c_regionkey,
    c_regionname,
    c_nationkey,
    c_nationname
)
SELECT * FROM query2;
```

```
WITH 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_mv
    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;
```

The total weight of the database with the materialized views is 19 GB (the constraint on the data warehouse size is respected).

3.5 Execution times

As for the queries with materialized views, we collected statistics on execution timings and results can be observed on Table 2. Further considerations are reported in the section 6.

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 queries 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 dimensions 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;
```

The total weight of the database with the indexes (without the materialized views) is 16 GB, in this way the bound given in section 1 is satisfied for the mandatory part of the project. What follows (i.e., indexes on materialized views) concerns optimizations that have been defined as *optional* in the project statement, and so it is not guaranteed that the limit remains fulfilled.

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. Results are reported in Table 3.

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	35740	34379	32891	32285	32067	33472	1556
2	55374	53992	53613	53575	55976	54506	1100
3	61	101	45	86	46	67	25

Table 3: Query timings with indexes, in milliseconds.

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.

```sql
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;

```

```sql
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;
```

The total weight of the database with the indexes on materialized views (which have been defined in section 3) is 23 GB (1.6 times the size of the initial database).

4.2.1 Execution times

The results of execution times for five independent runs on each query with all the aforementioned indexes (on tables and on materialized views) can be seen in Table 4.

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	22314	21998	21013	21356	20749	21486	658
2	24344	25531	22948	23063	24373	24052	1069
3	83	44	66	44	42	56	18

Table 4: Query timings with indexes on tables and materialized views, in milliseconds.

5 Fragmentation

To conclude the testings, it has been decided to try *vertical fragmentation*.

It makes sense to try such an approach, since queries only use a subset of attributes of the tables. *Vertical fragmentation* is usually implemented in distributes systems but it can still be useful in this case to reduce the workload induced by the queries.

```
---- NATION:
----- no fragmentation, the fragment used by the queries will have 3
    columns and the other 1 column only.

---- REGION:
----- no fragmentation, the fragment used by the queries will have 2
    columns and the other 1 column only.

---- CUSTOMER:
CREATE TABLE IF NOT EXISTS customer_frag_1
(
    c_custkey integer NOT NULL,
    c_name character varying(25) COLLATE pg_catalog."default" NOT
        NULL,
    c_nationkey integer NOT NULL,
    CONSTRAINT customer_frag_1_pkey PRIMARY KEY (c_custkey),
    CONSTRAINT customer_frag_1_fk1 FOREIGN KEY (c_nationkey)
        REFERENCES nation (n_nationkey) MATCH SIMPLE
        ON UPDATE NO ACTION
```

```
        ON DELETE NO ACTION
) AS
SELECT
    c_custkey,
    c_name,
    c_nationkey
FROM customer;
```

```
CREATE TABLE IF NOT EXISTS customer_frag_2
(
    c_custkey integer NOT NULL,
    c_address character varying(40) COLLATE pg_catalog."default" NOT
        NULL,
    c_phone character(15) COLLATE pg_catalog."default" NOT NULL,
    c_acctbal numeric(15,2) NOT NULL,
    c_mktsegment character(10) COLLATE pg_catalog."default" NOT NULL,
    c_comment character varying(117) COLLATE pg_catalog."default" NOT
        NULL,
    CONSTRAINT customer_frag_2_pkey PRIMARY KEY (c_custkey)
) AS
SELECT
    c_custkey,
    c_address,
    c_phone,
    c_acctbal,
    c_mktsegment,
    c_comment
FROM customer;
```

```
---- SUPPLIER:
CREATE TABLE IF NOT EXISTS supplier_frag_1
(
    s_suppkey integer NOT NULL,
    s_name character(25) COLLATE pg_catalog."default" NOT NULL,
    s_nationkey integer NOT NULL,
    CONSTRAINT supplier_frag_1_pkey PRIMARY KEY (s_suppkey),
    CONSTRAINT supplier_frag_1_fk1 FOREIGN KEY (s_nationkey)
        REFERENCES nation (n_nationkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
) AS
SELECT
```

```
    s_suppkey,
    s_name,
    s_nationkey
FROM supplier;
```

```
CREATE TABLE IF NOT EXISTS supplier_frag_2
(
    s_suppkey integer NOT NULL,
    s_address character varying(40) COLLATE pg_catalog."default" NOT
       NULL,
    s_phone character(15) COLLATE pg_catalog."default" NOT NULL,
    s_acctbal numeric(15,2) NOT NULL,
    s_comment character varying(101) COLLATE pg_catalog."default" NOT
       NULL,
    CONSTRAINT supplier_frag_2_pkey PRIMARY KEY (s_suppkey)
) AS
SELECT
    s_suppkey,
    s_address,
    s_phone,
    s_acctbal,
    s_comment
FROM supplier;

---- PARTSUPP: no fragmentation (table not used in queries)
```

```
---- PART:
CREATE TABLE IF NOT EXISTS part_frag_1
(
    p_partkey integer NOT NULL,
    p_type character varying(25) COLLATE pg_catalog."default" NOT
       NULL,
    CONSTRAINT part_frag_1_pkey PRIMARY KEY (p_partkey)
) AS
SELECT
    p_partkey,
    p_type
FROM part;
```

```
CREATE TABLE IF NOT EXISTS part_frag_2
(
    p_partkey integer NOT NULL,
```

```
    p_name character varying(55) COLLATE pg_catalog."default" NOT
        NULL,
    p_mfgr character(25) COLLATE pg_catalog."default" NOT NULL,
    p_brand character(10) COLLATE pg_catalog."default" NOT NULL,
    p_size integer NOT NULL,
    p_container character(10) COLLATE pg_catalog."default" NOT NULL,
    p_retailprice numeric(15,2) NOT NULL,
    p_comment character varying(23) COLLATE pg_catalog."default" NOT
        NULL,
    CONSTRAINT part_frag_2_pkey PRIMARY KEY (p_partkey)
) AS
SELECT
    p_partkey,
    p_name,
    p_mfgr,
    p_brand,
    p_size,
    p_container,
    p_retailprice,
    p_comment
FROM part;
```

```
---- ORDERS:
CREATE TABLE IF NOT EXISTS orders_frag_1
(
    o_orderkey integer NOT NULL,
    o_custkey integer NOT NULL,
    o_orderdate date NOT NULL,
    CONSTRAINT orders_frag_1_pkey PRIMARY KEY (o_orderkey),
    CONSTRAINT orders_frag_1_fk1 FOREIGN KEY (o_custkey)
        REFERENCES customer (c_custkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
) AS
SELECT
    o_orderkey,
    o_custkey,
    o_orderdate
FROM orders;
```

```
CREATE TABLE IF NOT EXISTS orders_frag_2
(
```

```
    o_orderkey integer NOT NULL,
    o_orderstatus character(1) COLLATE pg_catalog."default" NOT NULL,
    o_totalprice numeric(15,2) NOT NULL,
    o_orderpriority character(15) COLLATE pg_catalog."default" NOT
        NULL,
    o_clerk character(15) COLLATE pg_catalog."default" NOT NULL,
    o_shippriority integer NOT NULL,
    o_comment character varying(79) COLLATE pg_catalog."default" NOT
        NULL,
    CONSTRAINT orders_frag_2_pkey PRIMARY KEY (o_orderkey)
) AS
SELECT
    o_orderkey,
    o_orderstatus,
    o_totalprice,
    o_orderpriority,
    o_clerk,
    o_shippriority,
    o_comment
FROM orders;
```

```
---- LINEITEM:
CREATE TABLE IF NOT EXISTS lineitem_frag_1
(
    l_orderkey integer NOT NULL,
    l_partkey integer NOT NULL,
    l_suppkey integer NOT NULL,
    l_linenumber integer NOT NULL,
    l_extendedprice numeric(15,2) NOT NULL,
    l_discount numeric(15,2) NOT NULL,
    l_returnflag character(1) COLLATE pg_catalog."default" NOT NULL,
    l_commitdate date NOT NULL,
    l_receiptdate date NOT NULL,
    CONSTRAINT lineitem_frag_1_pkey PRIMARY KEY (l_orderkey,
        l_linenumber),
    CONSTRAINT lineitem_frag_1_fk1 FOREIGN KEY (l_orderkey)
        REFERENCES orders (o_orderkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT lineitem_frag_1_fk2 FOREIGN KEY (l_partkey, l_suppkey)
        REFERENCES partsupp (ps_partkey, ps_suppkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
```

```
) AS
SELECT
    l_orderkey,
    l_linenumber,
    l_partkey,
    l_suppkey,
    l_extendedprice,
    l_discount,
    l_returnflag,
    l_commitdate,
    l_receiptdate
FROM lineitem;
```

```
CREATE TABLE IF NOT EXISTS lineitem_frag_2
(
    l_orderkey integer NOT NULL,
    l_linenumber integer NOT NULL,
    l_quantity numeric(15,2) NOT NULL,
    l_tax numeric(15,2) NOT NULL,
    l_linestatus character(1) COLLATE pg_catalog."default" NOT NULL,
    l_shipdate date NOT NULL,
    l_shipinstruct character(25) COLLATE pg_catalog."default" NOT
        NULL,
    l_shipmode character(10) COLLATE pg_catalog."default" NOT NULL,
    l_comment character varying(44) COLLATE pg_catalog."default" NOT
        NULL,
    CONSTRAINT lineitem_frag_2_pkey PRIMARY KEY (l_orderkey,
        l_linenumber),
    CONSTRAINT lineitem_frag_2_fk FOREIGN KEY (l_orderkey)
        REFERENCES orders (o_orderkey) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
) AS
SELECT
    l_orderkey,
    l_linenumber,
    l_quantity,
    l_tax,
    l_linestatus,
    l_shipdate,
    l_shipinstruct,
    l_shipmode,
    l_comment
```

```
28 FROM lineitem;
```

The weight of the data warehouse with fragmented tables is roughly the same as the original one, since no additional data structures have been defined and the only action performed is a physical *split* of relations.

5.0.1 Execution times

Timings have been calculated using the queries defined in section 2 by only changing tables names.

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	14977	15803	15851	16517	16047	15839	558
2	20368	20474	19809	20284	20866	20360	380
3	2478	2150	2145	2350	2346	2294	147

Table 5: Query timings using fragmentation, in milliseconds.

5.1 Indexes on fragmented tables

Since the results shown in Table 5 are promising, it has been decided to implement the *indexes* (the ones defined in subsection 4.1) on the corresponding fragments.

The total size of the data warehouse at this point is 15 GB (again, the size constraint defined in section 1 is respected).

5.1.1 Execution times

Query	Run 1	Run 2	Run 3	Run 4	Run 5	μ	σ
1	24459	22156	21858	21829	21471	22355	1201
2	45378	40845	41757	40048	40449	41695	2154
3	143	164	61	67	93	106	46

Table 6: Query timings using fragmentation and indexes, in milliseconds.

6 Conclusions

Considering the overall results shown in Figure 1, the optimal approach for optimising the efficiency of queries 1 (Export/import revenue value) and 2 (Late delivery) might be to use the materialized views proposed in section 3 but this happens to be the worst-case scenario for query 3 (Returned item loss). The opposite situation occurs when using the indexes defined in section 4: the query 3 is optimised, but queries 1 and 2 show roughly the same run times as the naïve solution.

Assuming that all the three queries have the same importance (i.e., none of them is being executed a lot more frequently than the others), a good trade-off may seem to use *materialized views* and *indexes* (subsection 4.2). This solution, by the way, does not meet the size constraint of the project.

The final solution that is being proposed for the given problem concerns using the (*vertical*) *fragmentation* reported in section 5 without any additional index. Query 1 is executed in 15.84 s, i.e. 2.5 times faster than the naïve solution; query 2 runs in 20.36 s, i.e. it is 2.3 times faster than the related naïve solution; query 3 is executed in 2.29 s, i.e. 3.5 times faster than the corresponding naïve solution.

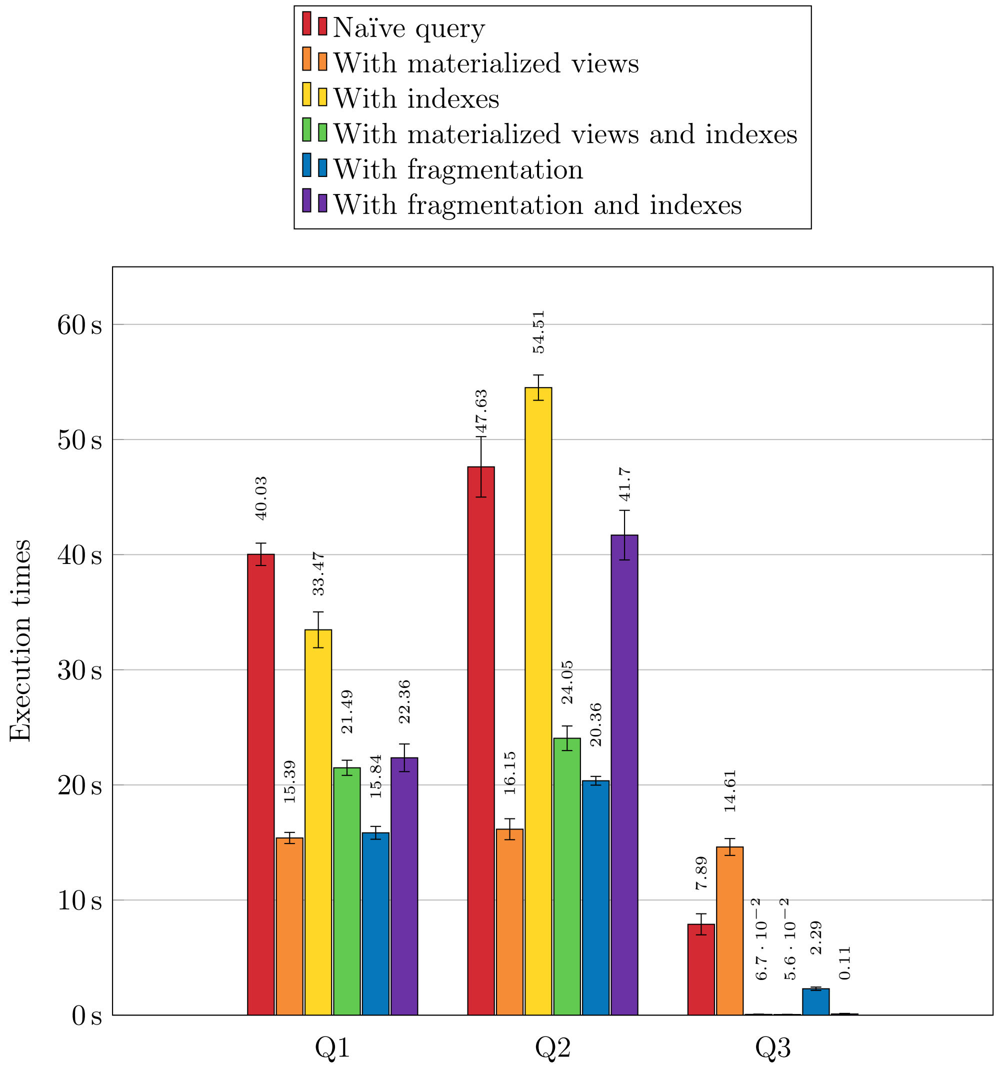

Figure 1: Query timings

References

[1] Transaction Processing Performance Council (TPC). *TPC BENCHMARK™ H (Decision Support) Standard Specification*. 2022. URL: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf.