DATA MANAGEMENT FOR BIG DATA PROJECT

Alessandro Cesa, Gabriel Costanzo

July 2023

Contents

Introduction

The exercise for the Data Management for Big Data exam focused on implementing and optimizing two specific query schemas on the TPC-H benchmark using PostgreSQL: Query Schema 1, which analyzes export/import revenue values, and Query Schema 3, which calculates returned item losses. These query schemas involved complex aggregations and calculations performed on large datasets, aiming to provide valuable insights into the company's operations.

The project's optimization strategy involved analyzing the query cost before optimization and comparing it with the costs after applying indexes and materialized views. The benefits of indexing and materialization in terms of query performance were assessed separately and in combination.

In the optimization process of the TPC-H benchmark project, we employed various techniques to enhance the performance of Query Schema 1 and Query Schema 3. To improve query execution, indexing was utilized to expedite data retrieval and reduce the need for full table scans. Indexes were designed based on the attributes involved in filtering, grouping, and joining operations. These indexes facilitated efficient access to the relevant data, resulting in faster query processing.

Furthermore, materialized views were employed to precompute and store the results of complex queries. By creating materialized views, we reduced the need to recompute the same aggregations repeatedly, improving query response times. Materialized views served as a form of data caching, allowing subsequent queries to retrieve results directly from the precomputed views.

To further enhance the performance of the materialized views, we also applied indexing to them. By designing appropriate indexes on the materi-

alized views.

By combining indexing, materialized views, and materialized views with indexes, we significantly optimized the execution of both Query Schema 1 and Query Schema 3. These techniques minimized query costs, reduced query execution time, and improved overall system performance. The strategic use of indexing and materialization allowed for faster data retrieval, reduced computational overhead and enhanced the efficiency of analytical operations in the TPC-H benchmark project.

The Database

The database is the TPC-H benchamrk database, a DB storing business-related informations about orders exchanging different items between various suppliers and customers located in different countries.

SQL definition of the tables

```
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 ( 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);
```

Tables population

We use command line of of pgAdmin4 to populate the tables with the following lines:

```
\copy customer FROM 'C:/Users/...../TPC-H V3.0.1/dbgen/customer.tbl'
WITH (FORMAT text, DELIMITER '|');
```

In this example we were populating the table "Customer"

Statistics of the Database

Size of tables

Table name	**Size**
LINEITEM	8788 MB
ORDERS	2039 MB
PARTSUPP	1363 MB
PART	320 MB
CUSTOMER	280 MB
SUPPLIER	17 MB
NATION	8,19 KB
REGION	8,19 KB
TOTAL	12,8 GB

Number of rows by table

Table name	**Number of rows**
LINEITEM	59986182
ORDERS	15000875
PARTSUPP	8000556
PART	2000037
CUSTOMER	1499999
SUPPLIER	100000
NATION	25
REGION	5

Statistics for attributes used in query schema 1

ATTRIBUTE	DIST_VALUES	MIN_VALUE	MAX_VALUE
TABLE: LINEITEM			
L_ORDERKEY	15000000	1	60000000
L_EXTENDEDPRICE	1351462	900.91	104949.50
L_SUPPKEY	100000	1	100000
L_DISCOUNT	11	0.00	0.00
L_PARTKEY	2000000	1	2000000
TABLE: ORDERS			
O_ORDERKEY	15000000	1	60000000
O_CUSTKEY	999982	1	1499999
O_ORDERDATE	2406	1992-01-01	1998-08-02
TABLE: CUSTOMER			
C_CUSTKEY	1500000	1	1500000
C_NATIONKEY	25	0	24
TABLE: SUPPLIER			
S_SUPPKEY	100000	1	100000
S_NATIONKEY	25	0	24
TABLE: PART			
P_PARTKEY	2000000	1	2000000
P_TYPE	150	—	—-
TABLE: NATION			
N_NATIONKEY	25	0	24
N_NAME	25	Algeria	Vietnam
N_REGIONKEY	5	0	4
TABLE: REGION			
R_NAME	5	Africa	Middle East
R_REGIONKEY	5	0	4

Statistics for attributes used in query schema 3

ATTRIBUTE	DIST_VALUES	MIN_VALUE	MAX_VALUE
TABLE: LINEITEM			
L_SHIPDATE	2526	1992-01-02	1998-12-01
L_EXTENDEDPRICE	1351462	900.91	104949.50
L_ORDERKEY	15000000	1	60000000
L_DISCOUNT	11	0.00	0.00
TABLE: ORDERS			
O_ORDERKEY	15000000	1	60000000
O_CUSTKEY	999982	1	1499999
TABLE: CUSTOMER			
C_CUSTKEY	1500000	1	1500000
C_NAME	1500000	Customer#000000001	Customer#001500000

Software and hardware

The Database Management System used for this project is PostgreSQL 15.3

We have used the Administration tool pgAdmin 4.

The Query Schema 1 was executed on a notebook HP 255 G6, with 4 GB of RAM and AMD® E2-9000e processor, and Ubuntu 22.04 operative system.

The Query Schema 3 was executed on a notebook DELL LATITUDE E7450, with 12 GB of RAM and INTEL(R) Core(TM) i7-5600U CPU@ 2.60GHz, and Windows 10 operative system.

Query Schema 1: Export/import revenue value

Problem statement

This Query Schema asks to compute aggregations 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 $\longrightarrow$ $Quarter \longrightarrow Year$
$Type$
$Nation \longrightarrow Region$
$TheslicingisoverTypeandExportingnation.$

Before Optimization

Query design and explanation

We need to compute aggregations of export/import revenues of lineitems between two different nations.

The aggregation levels are:

- **Q1**: Month,Nation
- **Q2**: Quarter,Nation
- **Q3**: Year,Nation

- **Q4**: Month,Region
- **Q5**: Quarter,Region
- **Q6**: Year,Region

We'll choose 'ECONOMY ANODIZED BRASS' for the slicing over Type and 'Algeria' for the slicing over the Exporting nation

We'll first conduct the basic queries without optimization

We'll select the needed attributes from the needed tables(LINEITEM, SUPPLIER,CUSTOMER,PART,NATION) and perform the joins. At this point we'll filter the rows where the exporting nation is different from the importing, we'll GROUP BY following the needed aggregation levels, ORDER in the needed order and select the requested atributes computing the revenue.

Queries

Q1:

```
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE*(1-SUB_LINEITEM.L_DISCOUNT))
AS REVENUE, E.N_NAME AS EXPORTING,I.N_NAME AS IMPORTING,
SUB_PART.P_TYPE AS TYPE,EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE)
AS MONTH, EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE) AS YEAR

FROM (SELECT L_ORDERKEY,L_PARTKEY,L_SUPPKEY,L_EXTENDEDPRICE,
L_DISCOUNT FROM LINEITEM) AS SUB_LINEITEM
JOIN (SELECT S_SUPPKEY,S_NATIONKEY FROM SUPPLIER) AS SUB_SUPPLIER
ON SUB_LINEITEM.L_SUPPKEY=SUB_SUPPLIER.S_SUPPKEY
JOIN (SELECT O_ORDERKEY,O_CUSTKEY,O_ORDERDATE FROM ORDERS)
AS SUB_ORDERS ON SUB_LINEITEM.L_ORDERKEY=SUB_ORDERS.O_ORDERKEY
JOIN (SELECT C_CUSTKEY,C_NATIONKEY FROM CUSTOMER) AS SUB_CUSTOMER
ON SUB_ORDERS.O_CUSTKEY=SUB_CUSTOMER.C_CUSTKEY
JOIN (SELECT P_PARTKEY,P_TYPE FROM PART WHERE P_TYPE='ECONOMY
ANODIZED BRASS') AS SUB_PART
ON SUB_LINEITEM.L_PARTKEY=SUB_PART.P_PARTKEY
JOIN (SELECT N_NATIONKEY,N_NAME
FROM NATION WHERE N_NAME='ALGERIA') AS E
ON SUB_SUPPLIER.S_NATIONKEY=E.N_NATIONKEY
JOIN (SELECT N_NATIONKEY,N_NAME FROM NATION)
AS I ON SUB_CUSTOMER.C_NATIONKEY=I.N_NATIONKEY

WHERE E.N_NATIONKEY!=I.N_NATIONKEY

GROUP BY E.N_NAME, I.N_NAME, SUB_PART.P_TYPE,
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)
```

```
ORDER BY E.N_NAME,I.N_NAME,SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE)
```

Q2:

```
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE*(1-SUB_LINEITEM.L_DISCOUNT))
AS REVENUE, E.N_NAME AS EXPORTING,I.N_NAME AS IMPORTING,
SUB_PART.P_TYPE AS TYPE, EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE)
AS QUARTER, EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE) AS YEAR

FROM (SELECT L_ORDERKEY,L_PARTKEY,L_SUPPKEY,L_EXTENDEDPRICE,
L_DISCOUNT FROM LINEITEM) AS SUB_LINEITEM
JOIN (SELECT S_SUPPKEY,S_NATIONKEY FROM SUPPLIER)
AS SUB_SUPPLIER  ON SUB_LINEITEM.L_SUPPKEY=SUB_SUPPLIER.S_SUPPKEY
JOIN (SELECT O_ORDERKEY,O_CUSTKEY,O_ORDERDATE FROM ORDERS)
AS SUB_ORDERS  ON SUB_LINEITEM.L_ORDERKEY=SUB_ORDERS.O_ORDERKEY
JOIN (SELECT C_CUSTKEY,C_NATIONKEY FROM CUSTOMER) AS SUB_CUSTOMER
ON SUB_ORDERS.O_CUSTKEY=SUB_CUSTOMER.C_CUSTKEY
JOIN (SELECT P_PARTKEY,P_TYPE FROM PART
WHERE P_TYPE='ECONOMY ANODIZED BRASS')
AS SUB_PART ON SUB_LINEITEM.L_PARTKEY=SUB_PART.P_PARTKEY
JOIN (SELECT N_NATIONKEY,N_NAME FROM NATION WHERE N_NAME='ALGERIA')
AS E ON SUB_SUPPLIER.S_NATIONKEY=E.N_NATIONKEY
JOIN (SELECT N_NATIONKEY,N_NAME FROM NATION)
AS I ON SUB_CUSTOMER.C_NATIONKEY=I.N_NATIONKEY

WHERE E.N_NATIONKEY!=I.N_NATIONKEY

GROUP BY E.N_NAME, I.N_NAME, SUB_PART.P_TYPE,
EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)

ORDER BY E.N_NAME,I.N_NAME,SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE)
```

Q3:

```
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE*(1-SUB_LINEITEM.L_DISCOUNT))
AS REVENUE, E.N_NAME AS EXPORTING,I.N_NAME AS IMPORTING,
SUB_PART.P_TYPE AS TYPE, EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)
AS YEAR

FROM (SELECT L_ORDERKEY,L_PARTKEY,L_SUPPKEY,L_EXTENDEDPRICE,
L_DISCOUNT FROM LINEITEM) AS SUB_LINEITEM
JOIN (SELECT S_SUPPKEY,S_NATIONKEY FROM SUPPLIER)
AS SUB_SUPPLIER  ON SUB_LINEITEM.L_SUPPKEY=SUB_SUPPLIER.S_SUPPKEY
JOIN (SELECT O_ORDERKEY,O_CUSTKEY,O_ORDERDATE FROM ORDERS)
AS SUB_ORDERS  ON SUB_LINEITEM.L_ORDERKEY=SUB_ORDERS.O_ORDERKEY
JOIN (SELECT C_CUSTKEY,C_NATIONKEY FROM CUSTOMER) AS SUB_CUSTOMER
ON SUB_ORDERS.O_CUSTKEY=SUB_CUSTOMER.C_CUSTKEY
JOIN (SELECT P_PARTKEY,P_TYPE FROM PART WHERE P_TYPE='ECONOMY
 ANODIZED BRASS') AS SUB_PART ON
 SUB_LINEITEM.L_PARTKEY=SUB_PART.P_PARTKEY
JOIN (SELECT N_NATIONKEY,N_NAME FROM NATION WHERE N_NAME='ALGERIA')
AS E  ON SUB_SUPPLIER.S_NATIONKEY=E.N_NATIONKEY
JOIN (SELECT N_NATIONKEY,N_NAME FROM NATION)
AS I ON SUB_CUSTOMER.C_NATIONKEY=I.N_NATIONKEY

WHERE E.N_NATIONKEY!=I.N_NATIONKEY

GROUP BY E.N_NAME, I.N_NAME, SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)

ORDER BY E.N_NAME,I.N_NAME,SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)
```

Q4:

```
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE*(1-SUB_LINEITEM.L_DISCOUNT))
AS REVENUE, E_REGION.R_NAME AS EXPORTING_REGION,
I_REGION.R_NAME AS IMPORTING_REGION,SUB_PART.P_TYPE AS TYPE,
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE) AS MONTH,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE) AS YEAR

FROM (SELECT L_ORDERKEY,L_PARTKEY,L_SUPPKEY,L_EXTENDEDPRICE,L_DISCOUNT
FROM LINEITEM) AS SUB_LINEITEM
JOIN (SELECT S_SUPPKEY,S_NATIONKEY FROM SUPPLIER) AS SUB_SUPPLIER
ON SUB_LINEITEM.L_SUPPKEY=SUB_SUPPLIER.S_SUPPKEY
JOIN (SELECT O_ORDERKEY,O_CUSTKEY,O_ORDERDATE FROM ORDERS)
AS SUB_ORDERS ON SUB_LINEITEM.L_ORDERKEY=SUB_ORDERS.O_ORDERKEY
JOIN (SELECT C_CUSTKEY,C_NATIONKEY FROM CUSTOMER) AS SUB_CUSTOMER
ON SUB_ORDERS.O_CUSTKEY=SUB_CUSTOMER.C_CUSTKEY
JOIN (SELECT P_PARTKEY,P_TYPE FROM PART
WHERE P_TYPE='ECONOMY ANODIZED BRASS')
AS SUB_PART ON SUB_LINEITEM.L_PARTKEY=SUB_PART.P_PARTKEY
JOIN (SELECT N_NATIONKEY,N_REGIONKEY FROM NATION) AS E
ON SUB_SUPPLIER.S_NATIONKEY=E.N_NATIONKEY
JOIN (SELECT R_REGIONKEY,R_NAME FROM REGION WHERE R_NAME='AFRICA')
AS E_REGION ON E.N_REGIONKEY=E_REGION.R_REGIONKEY
JOIN (SELECT N_NATIONKEY,N_REGIONKEY FROM NATION)
AS I ON SUB_CUSTOMER.C_NATIONKEY=I.N_NATIONKEY
JOIN (SELECT R_REGIONKEY,R_NAME FROM REGION)
AS I_REGION ON I.N_REGIONKEY=I_REGION.R_REGIONKEY

WHERE E_REGION.R_REGIONKEY!=I_REGION.R_REGIONKEY

GROUP BY E_REGION.R_NAME, I_REGION.R_NAME, SUB_PART.P_TYPE,
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)

ORDER BY E_REGION.R_NAME, I_REGION.R_NAME,SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE)
```

Q5:

```
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE * (1 - SUB_LINEITEM.L_DISCOUNT)) AS
    REVENUE,
       E_REGION.R_NAME AS EXPORTING_REGION,
       I_REGION.R_NAME AS IMPORTING_REGION,
       SUB_PART.P_TYPE AS TYPE,
       EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE) AS QUARTER,
       EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE) AS YEAR
FROM
  (SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_EXTENDEDPRICE, L_DISCOUNT
   FROM LINEITEM) AS SUB_LINEITEM
JOIN
  (SELECT S_SUPPKEY, S_NATIONKEY
   FROM SUPPLIER) AS SUB_SUPPLIER ON SUB_LINEITEM.L_SUPPKEY = SUB_SUPPLIER.
    S_SUPPKEY
JOIN
  (SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERDATE
   FROM ORDERS) AS SUB_ORDERS ON SUB_LINEITEM.L_ORDERKEY = SUB_ORDERS.
    O_ORDERKEY
JOIN
  (SELECT C_CUSTKEY, C_NATIONKEY
   FROM CUSTOMER) AS SUB_CUSTOMER ON SUB_ORDERS.O_CUSTKEY = SUB_CUSTOMER.
    C_CUSTKEY
JOIN
  (SELECT P_PARTKEY, P_TYPE
   FROM PART WHERE P_TYPE = 'ECONOMY ANODIZED BRASS') AS SUB_PART ON
    SUB_LINEITEM.L_PARTKEY = SUB_PART.P_PARTKEY
JOIN
  (SELECT N_NATIONKEY, N_REGIONKEY
   FROM NATION) AS E ON SUB_SUPPLIER.S_NATIONKEY = E.N_NATIONKEY
JOIN
  (SELECT R_REGIONKEY, R_NAME
   FROM REGION WHERE R_NAME = 'AFRICA') AS E_REGION ON E.N_REGIONKEY =
    E_REGION.R_REGIONKEY
JOIN
  (SELECT N_NATIONKEY, N_REGIONKEY
   FROM NATION) AS I ON SUB_CUSTOMER.C_NATIONKEY = I.N_NATIONKEY
JOIN
  (SELECT R_REGIONKEY, R_NAME
   FROM REGION) AS I_REGION ON I.N_REGIONKEY = I_REGION.R_REGIONKEY
WHERE E_REGION.R_REGIONKEY != I_REGION.R_REGIONKEY
GROUP BY E_REGION.R_NAME, I_REGION.R_NAME, SUB_PART.P_TYPE,
         EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE),
         EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)
ORDER BY E_REGION.R_NAME, I_REGION.R_NAME, SUB_PART.P_TYPE,
         EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE),
         EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE)
```

Q6:

```
Q6:
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE*(1-SUB_LINEITEM.L_DISCOUNT))
AS REVENUE,E_REGION.R_NAME AS EXPORTING_REGION,I_REGION.R_NAME
AS IMPORTING_REGION,SUB_PART.P_TYPE AS TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE) AS YEAR

FROM (SELECT L_ORDERKEY,L_PARTKEY,L_SUPPKEY,
L_EXTENDEDPRICE, L_DISCOUNT FROM LINEITEM) AS SUB_LINEITEM
JOIN (SELECT S_SUPPKEY,S_NATIONKEY FROM SUPPLIER) AS SUB_SUPPLIER
ON SUB_LINEITEM.L_SUPPKEY=SUB_SUPPLIER.S_SUPPKEY
JOIN (SELECT O_ORDERKEY,O_CUSTKEY,O_ORDERDATE FROM ORDERS) AS SUB_ORDERS
ON SUB_LINEITEM.L_ORDERKEY=SUB_ORDERS.O_ORDERKEY
JOIN (SELECT C_CUSTKEY,C_NATIONKEY FROM CUSTOMER) AS SUB_CUSTOMER
ON SUB_ORDERS.O_CUSTKEY=SUB_CUSTOMER.C_CUSTKEY
JOIN (SELECT P_PARTKEY,P_TYPE FROM PART WHERE P_TYPE='ECONOMY ANODIZED BRASS
    ')
AS SUB_PART ON SUB_LINEITEM.L_PARTKEY=SUB_PART.P_PARTKEY
JOIN (SELECT N_NATIONKEY,N_REGIONKEY FROM NATION) AS E
ON SUB_SUPPLIER.S_NATIONKEY=E.N_NATIONKEY
JOIN (SELECT R_REGIONKEY,R_NAME FROM REGION WHERE R_NAME='AFRICA')
AS E_REGION ON E.N_REGIONKEY=E_REGION.R_REGIONKEY
JOIN (SELECT N_NATIONKEY,N_REGIONKEY FROM NATION)
AS I ON SUB_CUSTOMER.C_NATIONKEY=I.N_NATIONKEY
JOIN (SELECT R_REGIONKEY,R_NAME FROM REGION)
AS I_REGION ON I.N_REGIONKEY=I_REGION.R_REGIONKEY

WHERE E_REGION.R_REGIONKEY!=I_REGION.R_REGIONKEY

GROUP BY E_REGION.R_NAME, I_REGION.R_NAME, SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)

ORDER BY E_REGION.R_NAME, I_REGION.R_NAME,SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)
```

We have the following running times:

QUERY	**RUNNING TIME(seconds)**
Q1	204
Q2	221
Q3	223
Q4	261
Q5	295
Q6	243
TOTAL	1447

Indexes

In order to optimize the Query, we'll implement indexing over some columns; We have tried to create indexes over all the columns that are involved in joins, WHERE and GROUP BY, and we have kept only the ones that are actually used by the Query Execution Plan. So we have created indexes on the columns:L_PARTKEY,C_CUSTKEY,O_ORDERKEY,P_TYPE,S_NATIONKEY.

Q_INDEXES:

```
CREATE INDEX idx_l_partkey ON LINEITEM(L_PARTKEY);
CREATE INDEX idx_c_custkey ON CUSTOMER(C_CUSTKEY);
CREATE INDEX idx_o_orderkey ON ORDERS(O_ORDERKEY);
CREATE INDEX idx_p_type ON PART(P_TYPE);
CREATE INDEX idx_s_nationkey ON SUPPLIER(S_NATIONKEY);
```

Indexes space and time cost

Computing these indexes takes 401 seconds.

These indexes occupy the following sizes:

ITEM	**SIZE(MB)**
idx_l_partkey	430
idx_c_custkey	32
idx_o_orderkey	321
idx_p_type	14
idx_s_nationkey	0.7
TOTAL	797.7

At this point we can re-run the 6 queries on the indexed database

Using the indexes we have the folloing Running time:

QUERY	**RUNNING TIME(seconds)**
Q1	87
Q2	105
Q3	99
Q4	123
Q5	127
Q6	119
TOTAL	660

Materialized views

In order to further optimize the queries, we'll now use materilized views. We'll create a materialized view reporting the export/import of revenue of lineitems between two different nations (E,I) on each month, with the slicing on the exporting region. At this point in order to get Q1,Q2,Q3 we'll have to slice over the exporting nation and do the roll-ups; in order to get Q3,Q4,q5 we'll have to select the rows in which regions are different and do the roll-ups.

Q_MAT:

```
CREATE MATERIALIZED VIEW MAT AS
SELECT SUM(SUB_LINEITEM.L_EXTENDEDPRICE*(1-SUB_LINEITEM.L_DISCOUNT))
AS REVENUE,E_REGION.R_NAME AS EXPORTING_REGION,
I_REGION.R_NAME AS IMPORTING_REGION,
E_NATION.N_NAME AS EXPORTING_NATION,I_NATION.N_NAME AS IMPORTING_NATION,
SUB_PART.P_TYPE AS TYPE,
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE) AS MONTH,
EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE) AS QUARTER,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE) AS YEAR

FROM (SELECT L_ORDERKEY,L_PARTKEY,L_SUPPKEY,L_EXTENDEDPRICE,
L_DISCOUNT FROM LINEITEM) AS SUB_LINEITEM
JOIN (SELECT S_SUPPKEY,S_NATIONKEY FROM SUPPLIER) AS SUB_SUPPLIER
ON SUB_LINEITEM.L_SUPPKEY=SUB_SUPPLIER.S_SUPPKEY
JOIN (SELECT O_ORDERKEY,O_CUSTKEY,O_ORDERDATE FROM ORDERS) AS SUB_ORDERS
ON SUB_LINEITEM.L_ORDERKEY=SUB_ORDERS.O_ORDERKEY
JOIN (SELECT C_CUSTKEY,C_NATIONKEY FROM CUSTOMER) AS SUB_CUSTOMER
ON SUB_ORDERS.O_CUSTKEY=SUB_CUSTOMER.C_CUSTKEY
JOIN (SELECT P_PARTKEY,P_TYPE FROM PART WHERE P_TYPE='ECONOMY ANODIZED BRASS
    ')
AS SUB_PART ON SUB_LINEITEM.L_PARTKEY=SUB_PART.P_PARTKEY
JOIN (SELECT N_NATIONKEY,N_NAME,N_REGIONKEY FROM NATION) AS E_NATION
ON SUB_SUPPLIER.S_NATIONKEY=E_NATION.N_NATIONKEY
JOIN (SELECT R_REGIONKEY,R_NAME FROM REGION WHERE R_NAME='AFRICA')
AS E_REGION ON E_NATION.N_REGIONKEY=E_REGION.R_REGIONKEY
JOIN (SELECT N_NATIONKEY,N_NAME,N_REGIONKEY FROM NATION)
AS I_NATION ON SUB_CUSTOMER.C_NATIONKEY=I_NATION.N_NATIONKEY
JOIN (SELECT R_REGIONKEY,R_NAME FROM REGION)
AS I_REGION ON I_NATION.N_REGIONKEY=I_REGION.R_REGIONKEY

WHERE E_NATION.N_NATIONKEY!=I_NATION.N_NATIONKEY

GROUP BY E_REGION.R_NAME, I_REGION.R_NAME,E_NATION.N_NAME,
I_NATION.N_NAME,SUB_PART.P_TYPE,
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE),
```

```
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE)

ORDER BY E_REGION.R_NAME, I_REGION.R_NAME,E_NATION.N_NAME,
I_NATION.N_NAME,SUB_PART.P_TYPE,
EXTRACT(YEAR FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(QUARTER FROM SUB_ORDERS.O_ORDERDATE),
EXTRACT(MONTH FROM SUB_ORDERS.O_ORDERDATE)
```

So now the queries will become
Q1:

```
SELECT REVENUE,EXPORTING_NATION AS EXPORTING,
IMPORTING_NATION AS IMPORTING, TYPE,MONTH,YEAR
FROM MAT
WHERE EXPORTING_NATION='ALGERIA'
ORDER BY EXPORTING_NATION,IMPORTING_NATION,TYPE,YEAR,MONTH
```

Q2:

```
SELECT SUM(REVENUE),EXPORTING_NATION AS EXPORTING,
IMPORTING_NATION AS IMPORTING,TYPE,QUARTER,YEAR
FROM MAT
WHERE EXPORTING_NATION='ALGERIA'
GROUP BY EXPORTING_NATION,IMPORTING_NATION,TYPE,QUARTER,YEAR
ORDER BY EXPORTING_NATION,IMPORTING_NATION,TYPE,YEAR,QUARTER
```

Q3:

```
SELECT SUM(REVENUE),EXPORTING_NATION AS EXPORTING,
IMPORTING_NATION AS IMPORTING,TYPE,YEAR
FROM MAT
WHERE EXPORTING_NATION='ALGERIA'
GROUP BY EXPORTING_NATION,IMPORTING_NATION,TYPE,YEAR
ORDER BY EXPORTING_NATION,IMPORTING_NATION,TYPE,YEAR
```

Q4:

```
SELECT REVENUE,EXPORTING_REGION AS EXPORTING,
IMPORTING_REGION AS IMPORTING, TYPE,MONTH,YEAR
FROM MAT
WHERE EXPORTING_REGION!=IMPORTING_REGION
ORDER BY EXPORTING_REGION,IMPORTING_REGION,TYPE,YEAR,MONTH
```

Q5:

```
SELECT SUM(REVENUE),EXPORTING_REGION AS EXPORTING,
IMPORTING_REGION AS IMPORTING, TYPE,QUARTER,YEAR
FROM MAT
WHERE EXPORTING_REGION!=IMPORTING_REGION
GROUP BY EXPORTING_REGION,IMPORTING_REGION,TYPE,QUARTER,YEAR
ORDER BY EXPORTING_REGION,IMPORTING_REGION,TYPE,YEAR,QUARTER
```

Q6:

```
SELECT SUM(REVENUE),EXPORTING_REGION AS EXPORTING,
IMPORTING_REGION AS IMPORTING, TYPE,YEAR
FROM MAT
WHERE EXPORTING_REGION!=IMPORTING_REGION
GROUP BY EXPORTING_REGION,IMPORTING_REGION,TYPE,YEAR
ORDER BY EXPORTING_REGION,IMPORTING_REGION,TYPE,YEAR
```

Materialized view space and time cost

The size for this materilized view is 1.736MB, and computing it takes 259 seconds.

QUERY	**RUNNING TIME(seconds)**
Q1	0.33
Q2	0.289
Q3	0.324
Q4	0.363
Q5	0.264
Q6	0.219
TOTAL	1.699

Materialized views with indexing

Now we'll create the materialized view with the indexes, create indexes on the materialized views and run the queries on it.

The indexes useful for creating the materialized view the indexes are: Q_INDEX_FOR_MAT:

```
CREATE INDEX idx_l_partkey ON LINEITEM(L_PARTKEY);
CREATE INDEX idx_o_orderkey ON ORDERS(O_ORDERKEY);
CREATE INDEX idx_p_type ON PART(P_TYPE);
```

Creating these indexes costs 329 seconds, and now creating the materialized view costs 140 seconds.

The only useful index on the materialized is in the column EXPORTING_NATION:

```
CREATE INDEX idx\_exporting\_nation ON MAT(EXPORTING_NATION);
```

Materialized view and indexes space and time cost

Creating the first indexes costs 329 seconds, and now creating the materialized view costs 140 seconds.

Creating the index on the materilized view costs 0.154 seconds

The indexes and the materialized view have the following sizes:

ITEM	**SIZE(MB)**
idx_l_partkey	430
idx_o_orderkey	321
idx_p_type	14
idx_exporting_nation	0.18
TOTAL	765.18

The Running times become:

QUERY	**RUNNING TIME(seconds)**
Q1	0.282
Q2	0.32
Q3	0.312
Q4	0.395
Q5	0.218
Q6	0.389
TOTAL	1.916

Comparisons

We can compare the total times taken by the optimization levels:

Optimization level	**RUNNING TIME(seconds)**
BASIC	1447
INDEXING	797.7
MATERIALIZATION	1.699
INDEXING AND MATERIALIZATION	1.916

We can see that, while already indexing is able to halve the running times, materialization is the best optimization technique, which enables us to pass from more than 24 minutes of the basic query to a running time of fewer than 2 seconds (although creating the materialized view costs a little more than 4 minutes, time can be halved by using indexes).

QUERY	**EXTRA SPACE(MB)**
BASIC	0
INDEXING	797.7
MATERIALIZATION	1.736
INDEXING AND MATERIALIZATION	767

The extra space occupied is well below the given bound.

Conclusion

We might consider plain materialization as the best technique since the running time is the lowest and it also uses less than 2MB (which is negligible given the size of the database) of extra space.

Query Schema 3: Returned item loss

Problem statement

This Query Schema asks to compute aggregations of the revenue loss for customers who might be having problems with the parts that are shipped to them.
Revenue lost is obtained by $sum(l_extendedprice * (1 - l_discount))$ for all qualifying lineitems.
The aggregations should be performed with the following roll-up
$Month \longrightarrow Quarter \longrightarrow Year$
Customer
The slicing is over the name of a customer and a specific quarter.

So, w'ell have to do the queries with the following aggregation levels:

- **Q1**: Month,Customer
- **Q2**: Quarter,Customer
- **Q3**: Year,Customer
- **We'll choose 'Customer(hash)001361741' for the slicing over the customer name and 'Q1' for the slicing over the aggregation: quarter.**

Before Optimization

We'll first conduct the basic queries without optimization

Query design and explanation

The given query is designed to calculate the revenue loss and item loss for customers based on the TPC-H benchmark. The aggregations are performed with a roll-up on the dimensions: Month, Quarter, Year, and Customer in PostgreSQL.

Here are the steps in the query:

1. The subquery is defined to retrieve the necessary data for the aggregation. It selects the following columns:

    - `TO_CHAR(l_shipdate, 'MM') AS month`: This expression extracts the year and month from the `l_shipdate` column and formats it as 'MM', representing the month.
    - `TO_CHAR(l_shipdate, '"Q"Q') AS quarter`: This expression extracts the year and quarter from the `l_shipdate` column and formats it as 'Q', representing the quarter.
    - `TO_CHAR(l_shipdate, 'YYYY') AS year`: This expression extracts the year from the `l_shipdate` column, representing the year.
    - `c.c_custkey AS customer_id`: Retrieves the customer key from the customer table as `customer_id`.
    - `l.l_extendedprice * (1 - l.l_discount) AS revenue_lost`: Calculates the revenue lost for each line item by multiplying the `l_extendedprice` with `(1 - l_discount)`.
    - `1 AS item_loss`: Represents the count of line items, indicating the number of items contributing to the revenue loss.

2. The subquery performs the necessary joins to retrieve the required data. It joins the `customer`, `orders`, and `lineitem` tables based on their key relationships. This allows us to fetch the relevant information for each customer and their associated orders and line items.

3. The main query then uses the subquery to perform the roll-up aggregations. It selects the following columns:

    - `subquery.month`: Represents the month extracted from the `l_shipdate` column.
    - `subquery.quarter`: Represents the quarter extracted from the `l_shipdate` column.

- `subquery.year`: Represents the year extracted from the `l_shipdate` column.
- `customer_id`: Represents the customer ID.
- `COALESCE(SUM(revenue_lost), 0) AS revenue_lost`: Calculates the total revenue lost for each combination of year, quarter, month, and customer. `COALESCE` function is used to handle NULL values and replace them with 0.
- `COALESCE(SUM(item_loss), 0) AS item_loss`: Calculates the total count of line items for each combination of year, quarter, month, and customer. Again, `COALESCE` function is used to handle NULL values and replace them with 0.

The `COALESCE` function is used in the query to handle the case where there are no matching line items for a particular customer. In the context of this query, the `COALESCE` function ensures that if there are no line items (i.e., no revenue lost) for a customer, the result will display 0 instead of a NULL value. It provides a default value to be used when the sum of revenue lost is NULL. By using `COALESCE`, you can guarantee that the result set will always have a value for the `revenue_lost` column, even if it's 0, for every customer. This can be helpful for further analysis or reporting purposes where you want consistent data representation.

4. The `GROUP BY` clause is used to group the results based on the dimensions: month, quarter, year, and `customer_id`.

5. The `ORDER BY` clause is used to sort the results in ascending order of month, quarter, year, and `customer_id`.

By executing this query, you will obtain the revenue loss and item loss for customers, aggregated based on the roll-up dimensions specified, and sorted in the specified order.

Q1: Aggregation by Month and Customer

```
SELECT
subquery.month,
customer_id,
COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
 SELECT
```

```
 TO_CHAR(l_shipdate, 'YYYY-MM') AS month,
c.c_custkey AS customer_id,
 l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
 1 AS item_loss
 FROM
 customer c
 INNER JOIN
 orders o ON c.c_custkey = o.o_custkey
 INNER JOIN
 lineitem l ON o.o_orderkey = l.l_orderkey
) subquery
GROUP BY (
subquery.month,
customer_id)
ORDER BY
subquery.month,
customer_id;
```

Q2: Aggregation by Quarter and Customer

```
SELECT
subquery.quarter,
customer_id,
COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
 SELECT
TO_CHAR(l_shipdate, ' YYYY-"Q"Q') AS quarter,
 c.c_custkey AS customer_id,
 l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
 1 AS item_loss
 FROM
 customer c
 INNER JOIN
 orders o ON c.c_custkey = o.o_custkey
 INNER JOIN
 lineitem l ON o.o_orderkey = l.l_orderkey
) subquery
GROUP BY (
subquery.quarter,
customer_id)
ORDER BY
subquery.quarter,
customer_id;
```

Q3: Aggregation by Year and Customer

```
SELECT
subquery.year,
customer_id,
COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
```

```
COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
 SELECT
TO_CHAR(l_shipdate, 'YYYY') AS year,
 c.c_custkey AS customer_id,
 l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
 1 AS item_loss
 FROM
 customer c
 INNER JOIN
 orders o ON c.c_custkey = o.o_custkey
 INNER JOIN
 lineitem l ON o.o_orderkey = l.l_orderkey
) subquery
GROUP BY (
subquery.year,
customer_id)
ORDER BY
subquery.year,
customer_id;
```

We have the following running times:

QUERY	**RUNNING TIME(minutes, seconds)**
Q1	5 min 9 sec
Q2	5 min 3 sec
Q3	4 min 3 sec
TOTAL	14 min 15 sec

Queries with slicing

We'll choose 'Customer(hash)001361741' for the slicing over the customer name and 'Q1' for the slicing over the aggregation: quarter.

Q1 is an example of slicing implementation with the level of aggregation by Month and Customer

```
SELECT
subquery.month,
subquery.customer_id,
subquery.customer_name,
COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
 SELECT
 TO_CHAR(l_shipdate, 'YYYY-MM') AS month,
c.c_custkey AS customer_id,
 l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
 1 AS item_loss,
```

```
 c.c_name AS customer_name
 FROM
 customer c
 INNER JOIN
 orders o ON c.c_custkey = o.o_custkey
 INNER JOIN
 lineitem l ON o.o_orderkey = l.l_orderkey
  WHERE
    c.c_name = 'Customer#001361741'
    AND TO_CHAR(l_shipdate, '"Q"Q') = 'Q1'
) subquery
GROUP BY (
subquery.month,
subquery.customer_id,
subquery.customer_name)
ORDER BY
subquery.month,
subquery.customer_id;
```

Queries time cost

We have the following running times for queries with slicing:

QUERY	**RUNNING TIME(minutes, seconds)**
Q1	4 min 10 sec
Q2	3 min 2 sec
Q3	3 min 38 sec
TOTAL	10 min 50 sec

Indexes

To optimize the query's execution time, I considered adding an index on the columns used in the join conditions. These are the SQL statements to create the indexes:

```
-- Create index idx_customer_custkey on customer table
CREATE INDEX idx_customer_custkey ON customer (c_custkey);

-- Create index idx_orders_o_custkey on orders table
CREATE INDEX idx_orders_o_custkey ON orders (o_custkey);

-- Create index idx_orders_orderkey on orders table
CREATE INDEX idx_orders_orderkey ON orders (o_orderkey);

-- Create index idx_lineitem_l_orderkey on lineitem table
```

```
CREATE INDEX idx_lineitem_l_orderkey ON lineitem (l_orderkey);
```

Indexes space and time cost

Computing these indexes takes 1 minute 26 seconds.

These indexes occupy the following sizes:

CREATED INDEX	**SPACE COST(MB)**
idx-customer-custkey	32
idx-orders-o-custkey	120
idx-orders-orderkey	321
idx-lineitem-l-orderkey	742
TOTAL	1,215 MB (1,21GB)

Running time for the queries after indexing:

QUERY	**RUNNING TIME(minutes, seconds)**
Q1	5 min 13 sec
Q2	4 min 50 sec
Q3	3 min 43 sec
TOTAL	10 min 50 sec

As we can see the running time for queries with or without indexing is practically the same. If we see the Query Execution Plan of the queries Q1, Q2 and Q3, all of them use the index

$$"idx_customer_custkeyoncustomerc"$$

.

Indexes space and time cost

We have the following running times for queries with slicing and indexing:

QUERY	**RUNNING TIME(seconds, miliseconds)**
Q1	47 sec 441 msec
Q2	35 sec 415 msec
Q3	38 sec 114 msec
TOTAL	120 sec 970 msec

As we can see, in this case, the running time for queries with indexing and slicing is reduced significantly. If we see the Query Execution Plan of the queries Q1, Q2, and Q3, all of them use the index "idx-orders-o-custkey on orders", which makes sense since o-custkey connects the Orders table

with the Customer table which contains the column c-name with the names of the customers.

Materialized views

To optimize the query's execution time we create the following materialized views:

```
CREATE MATERIALIZED VIEW my_materialized_view_1 AS
SELECT
    subquery.month,
    customer_id,
    COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
    COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
    SELECT
        TO_CHAR(l_shipdate, 'YYYY-MM') AS month,
        c.c_custkey AS customer_id,
        l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
        1 AS item_loss
    FROM
        customer c
        INNER JOIN orders o ON c.c_custkey = o.o_custkey
        INNER JOIN lineitem l ON o.o_orderkey = l.l_orderkey
) subquery
GROUP BY (subquery.month, customer_id)
ORDER BY subquery.month, customer_id;
```

The materialized views 2 and 3 are the same, only it is changing the level of aggregation, for Q2 is a quarter, and for Q3 is the year.

Materialized view with slicing (Q1)

```
CREATE MATERIALIZED VIEW my_materialized_view_4 AS
SELECT
subquery.month,
subquery.customer_id,
subquery.customer_name,
COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
 SELECT
 TO_CHAR(l_shipdate, 'YYYY-MM') AS month,
c.c_custkey AS customer_id,
 l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
 1 AS item_loss,
 c.c_name AS customer_name
 FROM
```

```
 customer c
 INNER JOIN
 orders o ON c.c_custkey = o.o_custkey
 INNER JOIN
 lineitem l ON o.o_orderkey = l.l_orderkey
  WHERE
    c.c_name = 'Customer#001361741'
    AND TO_CHAR(l_shipdate, '"Q"Q') = 'Q1'
) subquery
GROUP BY ROLLUP(
subquery.month,
subquery.customer_id,
subquery.customer_name)
ORDER BY
subquery.month,
subquery.customer_id;
```

Materialized views space and time cost

Materialized views	**SPACE COST(MB)**	**CREATION TIME COST**
Materialized views-1 (Q1)	1749	11 min 9 secs
Materialized views-2 (Q2)	1034	10 min
Materialized views-3 (Q3)	355	22 min 24 secs
Materialized views-4 (Q1) slicing	16 KB	1 min 11 secs
TOTAL	3,138 MB (3,14GB)	44 min 54 secs

To check if the materialized view is using the previously created indexes we use the following query:

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'my_materialized_view_1';
```

We could confirm that the materialized view is not using the previously created indexes.

Quering the Materialized views

```
SELECT * FROM my_materialized_view_1;
SELECT * FROM my_materialized_view_2;
SELECT * FROM my_materialized_view_3;
SELECT * FROM my_materialized_view_4;
```

These are the running times for queries with materialized views:

QUERY	RUNNING TIME(seconds, miliseconds)
Materialized views-1 (Q1)	41 sec 624 msec
Materialized views-2 (Q2)	26 sec 345 msec
Materialized views-3 (Q3)	8 sec 691 msec
Materialized views-4 (Q1) slicing	146 msec
TOTAL	76 sec 660 msec

As we can see in the table, the running time for the queries is reduced greatly after the creation of the materialized views. For each query the time is reduced approximately in:

Q1:4,5 min

Q2: 4,7 min

Q3: 3,9 min

Q1(slicing): 47 secs 295 msecs

Materialized views with indexing

We try how the query's execution time of materialized views changes by adding indexes. Since we saw previously that of the created indexes, only those that use the column c_custkey from the tables customers are effectively used. We decide to create this index inside the materialized views.

```
CREATE MATERIALIZED VIEW my_materialized_view_5 AS
SELECT
    subquery.month,
    customer_id,
    COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
    COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
    SELECT
        TO_CHAR(l_shipdate, 'YYYY-MM') AS month,
        c.c_custkey AS customer_id,
        l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
        1 AS item_loss
    FROM
        customer c
        INNER JOIN orders o ON c.c_custkey = o.o_custkey
        INNER JOIN lineitem l ON o.o_orderkey = l.l_orderkey
) subquery
GROUP BY (subquery.month, customer_id)
ORDER BY subquery.month, customer_id;
CREATE INDEX idx_customer_custkey_MV ON
my_materialized_view_5 (customer_id);
```

The materialized views 6 and 7 are the same, only it is changing the level of aggregation, for Q2 is a quarter, and for Q3 is the year.

Materialized views with indexes including slicing

Previously, after querying with created indexes on the query by slicing and inspecting the query execution plan, we realize that this query where using two indexes:

Index Scan using idx_orders_o_custkey on orders and

Index Cond: (o_custkey = c.c_custkey)

Briefly

Index Scan: An index scan refers to the process of sequentially scanning the entries of an index to fetch the required data.

Index Condition (Index Cond): An index condition, refers to the process of using an index to selectively fetch specific rows based on the conditions specified in the query. It involves applying filter conditions directly on the index entries to identify the relevant data, avoiding the need to scan the entire index. This method is advantageous when the query contains conditions that can be matched directly against the index's entries, allowing for a more efficient data retrieval process.

Based on this previous knowledge we design the following materialized view for query with slicing and indexes:

```
CREATE MATERIALIZED VIEW my_materialized_view_8 AS
SELECT
subquery.month,
subquery.customer_id,
subquery.customer_name,
COALESCE(SUM(revenue_lost), 0) AS revenue_lost,
COALESCE(SUM(item_loss), 0) AS item_loss
FROM (
 SELECT
 TO_CHAR(l_shipdate, 'YYYY-MM') AS month,
c.c_custkey AS customer_id,
 l.l_extendedprice * (1 - l.l_discount) AS revenue_lost,
 1 AS item_loss,
```

```
 c.c_name AS customer_name
 FROM
 customer c
 INNER JOIN
 orders o ON c.c_custkey = o.o_custkey
 INNER JOIN
 lineitem l ON o.o_orderkey = l.l_orderkey
  WHERE
    c.c_name = 'Customer#001361741'
    AND TO_CHAR(l_shipdate, '"Q"Q') = 'Q1'
) subquery
GROUP BY ROLLUP(
subquery.month,
subquery.customer_id,
subquery.customer_name)
ORDER BY
subquery.month,
subquery.customer_id;
CREATE INDEX idx_customer_custkey_MV_8 ON my_materialized_view_8 (
    customer_id);
```

Materialized views with indexes, space, and time cost

Materialized views	**SPACE COST(MB)**	**CREATION TIME COST**
Materialized views-5 (Q1)	1966	8 min 32 secs
Materialized views-6 (Q2)	1160	7 min 7 secs
Materialized views-7 (Q3)	421	5 min 44 secs
Materialized views-8 (Q1) slicing	32 KB	1 sec 576 msecs
TOTAL	3,547 MB (3,55GB)	21 min 14 secs

Quering the Materialized views with indexes

```
SELECT * FROM my_materialized_view_5;
SELECT * FROM my_materialized_view_6;
SELECT * FROM my_materialized_view_7;
SELECT * FROM my_materialized_view_8;
```

These are the running times for queries with materialized views and indexes:

QUERY	**RUNNING TIME(seconds, miliseconds)**
Materialized views-5 (Q1)	32 sec 433 msec
Materialized views-6 (Q2)	18 sec 28 msec
Materialized views-7 (Q3)	6 sec 976 msec
Materialized views-8 (Q1) slicing	586 msec
TOTAL	58 sec

Comparisons

RUNTIME

QUERY	ORIGINAL	WITH INDEXES	WITH MV	MV-INDEX
Q1	5min 9sec	5min 13sec	41sec 624msec	32sec 433msec
Q2	5min 3sec	4min 50sec	26sec 345msec	18sec 28msec
Q3	4min 3sec	3min 43sec	8sec 691msec	6 sec 976msec
Q1 (with slicing)	4min 10sec	47sec 441 msec	146msec	586msec

Although we check in the query execution plan (QEP) that after the creation of the indexes, they are used in the execution of the queries, as we can see in the recap table, the addition of indexes has not had a significant impact on the queries runtimes except in the case of the queries that include slicing. As we can see in the table, in optimizing the runtime of the queries what makes the difference is the creation of materialized views (MV). The result is a large decrease in the runtime, from minutes to seconds. However, adding indexes to the MV showed no significant improvement in the runtime.

SPACE COST RECAP
Indexes: 1.21 GB
Materialized views: 3.14 GB
Materialized views and indexes: 3.55GB

Conclusion

Considering the obtained results and depending on the frequency of the queries, it should be recommended to add indexes if we have frequent queries with slicing. On the other hand, in general, the materialized view creation of frequent queries improves greatly the runtimes of the queries. Lastly, in making a cost-benefit analysis, incorporating indexes poorly improves the runtimes of the queries and is costly in terms of space, then it is not advisable to add indexes to the materialized views.