Data Warehouse - Assignment 5 - SWEN 432

Zoltan Debre - 300360191

Original repository and progress history: https://github.com/zoltan-nz/postgresql-datawarehouse-excercise

Everything inserted in one runnable `sql` file which can be run with the following way if `dbname` exists:

```
$ psql --dbname=zoltan --file=./assignment-5.sql
```

The above command will import also the `dump` file.

Question 1 - A Data Mart in the PostgreSQL Environment

(12 marks)

Create and populate `Book Orders Database` from `BookOrdersDatabaseDump_17.sql` .

```
$ createdb zoltan
$ psql -d zoltan -f ./BookOrdersDatabaseDump_17.sql
```

A little database cleanup, renaming all `Sidney` to `Sydney` .

```
UPDATE customer SET city = 'Sydney' WHERE customer.city = 'Sidney';
UPDATE customer SET district = 'Povardarje' WHERE CustomerId = 96;
UPDATE customer SET district = 'Budapest' WHERE CustomerId = 100;
```

Create `Data Mart` according to the description of its schema in Table 2.

Populate it by extracting, transforming, and loading data from the operational “Book Orders Database”. Use PostgreSQL commands and functions to accomplish the tasks.

When building Time dimension, note that:

- `TimeId` should be a sequence generated by PostgreSQL. That sequence is associated with the `Cust_Order.OrderDate` values in an ascending manner (the earliest `Cust_Order.OrderDate` date is associated with the `TimeId = 1`).
- There are various PostgreSQL functions that allow automatic transformation of `Cust_Order.OrderDate` values into appropriate surrogates of `DayOfWeek` , `Month` , and real `Year` values.

- The `DayOfWeek` attribute takes values from the set {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')}.
- The `Month` attribute takes values from the set {'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December'}.

Creating TIME table

```
DROP TABLE IF EXISTS time;

CREATE TABLE time
(
  TimeId    SERIAL PRIMARY KEY NOT NULL,
  OrderDate DATE               NOT NULL,
  DayOfWeek CHAR(10)           NOT NULL,
  Month     CHAR(10)           NOT NULL,
  Year      INT                NOT NULL
);
CREATE UNIQUE INDEX time_TimeId_uindex ON time (TimeId);
```

- Populate from `cust_order` table:

```
INSERT INTO time (OrderDate, DayOfWeek, Month, Year)
  SELECT DISTINCT
    cust_order.orderdate                    AS OrderDate,
    to_char(cust_order.orderdate, 'Day')    AS DayOfWeek,
    to_char(cust_order.orderdate, 'Month')  AS Month,
    extract(YEAR FROM cust_order.orderdate) AS Year
  FROM cust_order
  ORDER BY OrderDate ASC;
```

Result:

```
SELECT COUNT(time.TimeId) FROM time;
 count
-------
   124
(1 row)
```

Creating SALES table

Note: `Amnt = SUM(Order_Detail.Quantity * Book.Price)`

Traditional way. We crate a table and insert data from other tables. (Below we can see a more efficient way to create sales aggregation with materialized view. I will use the materialized view `sales` table in this

assignment.)

```
CREATE TABLE sales_table
(
  customerid INTEGER       NOT NULL
    CONSTRAINT sales_customer_customerid_fk
    REFERENCES customer,
  timeid     INTEGER       NOT NULL
    CONSTRAINT sales_time_timeid_fk
    REFERENCES time,
  isbn       INTEGER       NOT NULL
    CONSTRAINT sales_book_isbn_fk
    REFERENCES book,
  amnt       NUMERIC(6, 2) NOT NULL,
  CONSTRAINT sales_customerid_timeid_isbn_pk PRIMARY KEY (customerid, timeid,
    isbn)
);

INSERT INTO sales_table (customerid, timeid, isbn, amnt)
  SELECT
    customer.customerid                    AS customerid,
    time.timeid                            AS timeid,
    book.isbn                              AS isbn,
    sum(order_detail.quantity * book.price) AS amnt
  FROM order_detail NATURAL JOIN book NATURAL JOIN cust_order NATURAL JOIN
    customer NATURAL JOIN time
  GROUP BY customer.customerid, time.timeid, book.isbn;
```

I will use the following materialized view as `sales` fact table.

Creating SALES fact table (using materialized view)

Note: `Amnt = SUM(Order_Detail.Quantity * Book.Price)`

This is an alternative approach, so we can use materialized view for storing sales aggregation.

```
DROP MATERIALIZED VIEW IF EXISTS sales CASCADE;

CREATE MATERIALIZED VIEW sales AS
  SELECT
    customer.customerid                                          AS CustomerId,
    time.timeid                                                  AS TimeId,
    book.isbn                                                    AS ISBN,
    sum(order_detail.quantity * book.price) :: numeric(6, 2) AS Amnt
  FROM book NATURAL JOIN order_detail NATURAL JOIN cust_order NATURAL JOIN
    customer NATURAL JOIN time
  GROUP BY customer.customerid, time.timeid, book.isbn
  ORDER BY CustomerId, TimeId, ISBN;

CREATE UNIQUE INDEX sales_CustomerIdTimeIdISBN_uindex ON sales (CustomerId,
  TimeId, ISBN);
```

Result:

```
SELECT count(*) FROM sales;
 count
-------
  1070
(1 row)
```

Question 2 - Aggregate Queries

(14 marks)

Find the average amount of money spent by all customers on buying books for all days so far.

Calculating averages from averages are wrong approach, we will get wrong results as we can see in the following queries:

```
CREATE MATERIALIZED VIEW avg_amnt_view AS
  SELECT
    CustomerId,
    avg(Amnt) AS avg_amnt
  FROM sales
  GROUP BY customerid;

SELECT 104
SELECT avg(avg_amnt) FROM avg_amnt_view;
         avg
----------------------
 202.9588687852809865
(1 row)
```

In the following case we calculate the average of all individual transactions. A transaction is amount of spending per customer per day per book.isbn. First I thought, this is not what we are looking for, but I got a message from my classmates, that we are looking for "the average money spent per customer, per day, per book.isbn". In this case the following solution is right, because our sales table's each row is unique for this three factors.

```
SELECT avg(amnt) FROM sales;
         avg
----------------------
 161.3691588785046729
(1 row)
```

But I think "average amount of money spent by all customers on buying books for all days" should remove the `book.isbn` as a property, so should combine book sales by customer and by day. Using two dimensional aggregation for calculating the average. Because I calculated and implemented that solution also, I just leave it here below.

Altenative solution:

We are looking for the average amount of money spent by all customers for all days, so first we have to create a intermediate tuple. In this case each row represents a unique customer-day transaction. A customer can come back next day and could have a purchase, so it will be a unique row.

```
CREATE MATERIALIZED VIEW sum_customer_per_day AS
  SELECT
    customerid,
    timeid,
    sum(amnt) as amnt_spent_daily_by_customers
  FROM sales
  GROUP BY customerid, timeid;

SELECT 198

SELECT avg(amnt_spent_daily_by_customers) AS avg_spending_by_customers_per_day
  from sum_customer_per_day;

 avg_spending_by_customers_per_day
-----------------------------------
              872.0454545454545455
(1 row)
```

Or we can calculate from the other direction. First creating a materialized view which list the average spending by customer each day and using this avg and count to calculate our final daily avg spending.

```
CREATE MATERIALIZED VIEW avg_spending_by_customer_on_each_day AS
  SELECT
    timeid,
    count(customerid)
      AS number_of_customer_a_day,
    avg(sum_customer_per_day.amnt_spent_daily_by_customers)
      AS avg_spending FROM sum_customer_per_day
  GROUP BY timeid;

SELECT 124

SELECT
  sum(avg_spending_by_customer_on_each_day.avg_spending *
    avg_spending_by_customer_on_each_day.number_of_customer_a_day)
  / sum(avg_spending_by_customer_on_each_day.number_of_customer_a_day)
    AS Total_AVG
FROM avg_spending_by_customer_on_each_day;

      total_avg
----------------------
 872.0454545454545455
```

Question 3. OLAP Queries

(20 marks)

a)

Use SQL to retrieve from your Data Mart: `customer id`'s, `names` and `surnames` of `five` customers who spent the **largest** amount of money buying books. This query uses two OLAP specific operations, name them.

(The result added to a materialized view, because we use it later.)

```
CREATE MATERIALIZED VIEW best_buyers AS
SELECT
  customer.CustomerId AS customer_id,
  customer.f_name     AS first_name,
  customer.l_name     AS last_name,
  sum(amnt)           AS spending
FROM sales
  NATURAL JOIN customer
GROUP BY customer.CustomerId
ORDER BY spending DESC LIMIT 5;

SELECT 5

SELECT * FROM best_buyers;

 customer_id |      first_name      |      last_name       | spending
-------------+----------------------+----------------------+----------
           1 | Kirk                 | Jacson               | 17810.00
           3 | Peter                | Andree               | 14100.00
          14 | Craig                | Anslow               | 11780.00
           2 | May-N                | Leow                 |  7145.00
          79 | Jiajun               | Liang                |  6095.00
(5 rows)
```

Used OLAP operations:

- *Dimensional roll-ups*, because we drop `book` and `time` dimension from our calculation.
- *Pivoting*, because we use three dimensions to aggregate our values for our `sales` table what we use for our calculation to get our best buyers.

In terms of *Aggregate Functions*, we can classify our query as *Rank Query* and *Top N Query* also, because we rank them based on spending and we just select the top 5.

b)

Use SQL to find from your `Data Mart` and `the operational database` whether the customer who spent the greatest amount of money buying books did this by issuing many orders with smaller amounts or a few orders with greater amounts of money, or even great number of orders with greater amounts of money. Base your answer on an appropriate average value and the percentage of best buyer's orders being smaller or greater than this average. What is the name of that OLAP specific operation? (You may use a stepwise procedure to solve the question.)

- `ord_avg_amnt` : the average amount of money of all orders

The first materialized view calculate the value of orders and after we calculate the average value of this list. "`sql CREATE MATERIALIZED VIEW amountperorder AS SELECT orderdetail.orderid, sum(orderdetail.quantity * book.price) AS orderamount FROM orderdetail NATURAL JOIN book GROUP BY orderid;

SELECT 222

CREATE MATERIALIZED VIEW ordavgamnt AS SELECT avg(amountperorder.orderamount) AS ordavgamnt FROM amountper_order;

SELECT 1

SELECT * FROM ordavgamnt;

```
ord_avg_amnt
```

777.7702702702702703 (1 row) "`

Used OLAP operation: *Roll-up* - we use multiple aggregation and group by to generate the `amount_per_order` table.

- `no_of_ord` : the number of orders issued by the customer who spent the greatest amount of money buying books (the best buyer)

```
CREATE MATERIALIZED VIEW no_of_ord AS
  SELECT count(cust_order.orderid) AS no_of_ord FROM cust_order
  WHERE cust_order.customerid IN (SELECT customer_id from best_buyers limit 1)
  GROUP BY cust_order.customerid;

SELECT 1

SELECT * FROM no_of_ord;

 no_of_ord
-----------
        14
(1 row)
```

Used OLAP operation: *Slice* - number of dimensions are smaller and we use condition with WHERE clause.

- `perc_of_ord` : the percentage of orders issued by the best buyer that had a greater total amount than the ordavgamnt.

```
CREATE MATERIALIZED VIEW amount_per_order_by_customer AS
  SELECT
    order_detail.orderid,
    sum(order_detail.quantity * book.price) AS order_amount
  FROM order_detail NATURAL JOIN book NATURAL JOIN cust_order
    NATURAL JOIN customer
  WHERE cust_order.customerid IN (SELECT customer_id FROM best_buyers LIMIT 1)
  GROUP BY orderid;

SELECT 14
```

Used OLAP operation: *Roll-up* - we use aggregation and GROUP BY. *Slice* - we use WHERE with conditions to reduce dimensions.

List of orders by the customers with amount:

```
SELECT * FROM amount_per_order_by_customer;
 orderid | order_amount
---------+--------------
     170 |       250.00
     107 |      2535.00
     111 |       915.00
       8 |      1245.00
      19 |      1120.00
     108 |      4165.00
       1 |       885.00
      21 |       395.00
     112 |       260.00
       4 |       925.00
     110 |      1910.00
       5 |       925.00
     172 |       450.00
     114 |      1830.00
(14 rows)
```

How many percentage above average:

```
CREATE MATERIALIZED VIEW perc_of_ord AS
  SELECT (count(*) * 100) :: NUMERIC / no_of_ord.no_of_ord AS perc_of_ord
  FROM amount_per_order_by_customer NATURAL JOIN ord_avg_amnt
    NATURAL JOIN no_of_ord
  WHERE amount_per_order_by_customer.order_amount > ord_avg_amnt.ord_avg_amnt
  GROUP BY no_of_ord.no_of_ord;

SELECT 1

SELECT * FROM perc_of_ord;

     perc_of_ord
---------------------
 71.4285714285714286
(1 row)
```

Used OLAP operation: Roll-up - we use aggregation and GROUP BY. Slice - we use WHERE with conditions to reduce dimensions.

Conclusion:

```
SELECT perc_of_ord,
  CASE
    WHEN perc_of_ord >= 75
      THEN 'we estimate that the best buyer has issued a greater
       (than average) number of orders with greater (than average)
       amounts of money'
    WHEN perc_of_ord < 75 AND perc_of_ord >= 50
      THEN 'we estimate that the best buyer has issued a greater
       (than average) to medium number of orders with greater (than average)
       amounts of money'
    WHEN perc_of_ord < 50 AND perc_of_ord >= 25
      THEN 'we estimate that the best buyer has issued a small to medium
       number of orders with greater (than average) amounts of money'
    WHEN perc_of_ord < 25
      THEN 'we estimate that the best buyer has issued a small number
       of orders with greater (than average) amounts of money'
  END
FROM perc_of_ord;

     perc_of_ord     |                                          case
---------------------+-----------------------------------------------------------------
 71.4285714285714286 | we estimate that the best buyer has issued a greater
                     | (than average) to medium number of
                     | orders with greater (than average) amounts of money
(1 row)
```

Question 4 - Queries Against Materialized Views

(24 marks)

a)

Use SQL to materialize the following two views:

```
CREATE MATERIALIZED VIEW View1 AS
   SELECT c.CustomerId, F_Name, L_Name, District, TimeId,
   DayOfWeek, ISBN, Amnt
   FROM Sales NATURAL JOIN Customer c NATURAL JOIN Time;

CREATE MATERIALIZED VIEW View2 AS
   SELECT c.CustomerId, F_Name, L_Name, Year, SUM(Amnt)
   FROM Sales NATURAL JOIN Customer c NATURAL JOIN Time
   GROUP BY c.CustomerId, F_Name, L_Name, Year;
```

Use PostgreSQL `EXPLAIN ANALIZE` (and `VACUUM ANALYZE`) command to get time needed to

retrieve five best buyers of question 3.a) when the SQL statement is issued against: 1. The "Book Orders Database" 2. The Data Mart, 3. The view View1, and 4. The view View2.

The Book Orders Database

```
EXPLAIN ANALYZE
SELECT
  customer.CustomerId AS customer_id,
  customer.f_name     AS first_name,
  customer.l_name     AS last_name,
  sum(amnt)           AS spending
FROM
  (
    SELECT
      customer.customerid                                  AS CustomerId,
      time.timeid                                          AS TimeId,
      book.isbn                                            AS ISBN,
      sum(order_detail.quantity * book.price) :: NUMERIC(6, 2)
        AS Amnt
    FROM book NATURAL JOIN order_detail NATURAL JOIN cust_order
      NATURAL JOIN customer NATURAL JOIN time
    GROUP BY customer.customerid, time.timeid, book.isbn
    ORDER BY CustomerId, TimeId, ISBN
  ) AS sales
  NATURAL JOIN customer
GROUP BY customer.CustomerId
ORDER BY spending DESC LIMIT 5;
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=1029.47..1029.49 rows=5 width=78) (actual time=2.739..2.740 rows=5 loo
   ->  Sort  (cost=1029.47..1029.77 rows=118 width=78) (actual time=2.738..2.738 row
         Sort Key: (sum(((sum(((order_detail.quantity)::numeric * book.price)))::num
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=1026.04..1027.51 rows=118 width=78) (actual time=2
               Group Key: customer.customerid
               ->  Hash Join  (cost=654.78..1006.16 rows=3975 width=60) (actual time
                     Hash Cond: (customer_1.customerid = customer.customerid)
                     ->  GroupAggregate  (cost=646.13..865.11 rows=6738 width=26) (a
                           Group Key: customer_1.customerid, "time".timeid, book.isb
                           ->  Sort  (cost=646.13..662.97 rows=6738 width=19) (actua
                                 Sort Key: customer_1.customerid, "time".timeid, boo
                                 Sort Method: quicksort  Memory: 134kB
                                 ->  Hash Join  (cost=85.80..217.65 rows=6738 width=
                                       Hash Cond: (order_detail.orderid = cust_order
                                       ->  Hash Join  (cost=1.27..60.99 rows=1925 wi
                                             Hash Cond: (order_detail.isbn = book.is
                                             ->  Seq Scan on order_detail  (cost=0.0
```

```
                                        ->  Hash  (cost=1.12..1.12 rows=12 widt
                                              Buckets: 1024  Batches: 1  Memory
                                              ->  Seq Scan on book  (cost=0.00.
                                  ->  Hash  (cost=69.95..69.95 rows=1166 width=
                                        Buckets: 2048  Batches: 1  Memory Usage
                                        ->  Hash Join  (cost=22.92..69.95 rows=
                                              Hash Cond: ("time".orderdate = cu
                                              ->  Seq Scan on "time"  (cost=0.0
                                              ->  Hash  (cost=18.75..18.75 rows
                                                    Buckets: 1024  Batches: 1
                                                    ->  Hash Join  (cost=8.65..
                                                          Hash Cond: (cust_orde
                                                          ->  Seq Scan on cust_
                                                          ->  Hash  (cost=7.18.
                                                                Buckets: 1024
                                                                ->  Seq Scan or
                        ->  Hash  (cost=7.18..7.18 rows=118 width=46) (actual time=0.04
                              Buckets: 1024  Batches: 1  Memory Usage: 17kB
                              ->  Seq Scan on customer  (cost=0.00..7.18 rows=118 width
 Planning time: 0.360 ms
 Execution time: 2.793 ms
(39 rows)

VACUUM ANALYZE;
VACUUM
```

21 loops

The Data Mart

```
EXPLAIN ANALYZE
SELECT
  customer.CustomerId AS customer_id,
  customer.f_name     AS first_name,
  customer.l_name     AS last_name,
  sum(amnt)           AS spending
FROM sales
  NATURAL JOIN customer
GROUP BY customer.CustomerId
ORDER BY spending DESC LIMIT 5;
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------
 Limit  (cost=49.85..49.86 rows=5 width=78) (actual time=0.690..0.691 rows=5 loops=1
   ->  Sort  (cost=49.85..50.15 rows=118 width=78) (actual time=0.690..0.691 rows=5
         Sort Key: (sum(sales.amnt)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=46.42..47.89 rows=118 width=78) (actual time=0.637
               Group Key: customer.customerid
               ->  Hash Join  (cost=8.65..41.07 rows=1070 width=51) (actual time=0.0
                     Hash Cond: (sales.customerid = customer.customerid)
                     ->  Seq Scan on sales  (cost=0.00..17.70 rows=1070 width=9) (ac
                     ->  Hash  (cost=7.18..7.18 rows=118 width=46) (actual time=0.03
                           Buckets: 1024  Batches: 1  Memory Usage: 17kB
                           ->  Seq Scan on customer  (cost=0.00..7.18 rows=118 width
 Planning time: 0.251 ms
 Execution time: 0.726 ms
(14 rows)

VACUUM ANALYZE;
VACUUM
```

7 loops

The View1

```
EXPLAIN ANALYZE
SELECT
  customerid AS customer_id,
  f_name     AS first_name,
  l_name     AS last_name,
  sum(amnt)  AS spending
FROM View1
GROUP BY customer_id, first_name, last_name
ORDER BY spending DESC LIMIT 5;
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------
 Limit  (cost=41.51..41.53 rows=5 width=78) (actual time=0.699..0.700 rows=5 loops=1
   ->  Sort  (cost=41.51..41.78 rows=107 width=78) (actual time=0.698..0.699 rows=5
         Sort Key: (sum(amnt)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=38.40..39.74 rows=107 width=78) (actual time=0.647
               Group Key: customerid, f_name, l_name
               ->  Seq Scan on view1  (cost=0.00..27.70 rows=1070 width=51) (actual
 Planning time: 0.099 ms
 Execution time: 0.725 ms
(9 rows)

VACUUM ANALYZE;
VACUUM
```

4 loops

The View2

```
EXPLAIN ANALYZE
SELECT
  customerid AS customer_id,
  f_name     AS first_name,
  l_name     AS last_name,
  sum(sum)   AS spending
FROM View2
GROUP BY customer_id, first_name, last_name
ORDER BY spending DESC LIMIT 5;
                                                              QUERY PLAN
--------------------------------------------------------------------------------------
 Limit  (cost=7.67..7.68 rows=5 width=78) (actual time=0.138..0.139 rows=5 loops=1)
   ->  Sort  (cost=7.67..7.93 rows=104 width=78) (actual time=0.138..0.139 rows=5 lo
         Sort Key: (sum(sum)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=4.64..5.94 rows=104 width=78) (actual time=0.085..
               Group Key: customerid, f_name, l_name
               ->  Seq Scan on view2  (cost=0.00..3.32 rows=132 width=51) (actual ti
 Planning time: 0.087 ms
 Execution time: 0.163 ms
(9 rows)

VACUUM ANALYZE;
VACUUM
```

4 loops

Findings:

- We need less run loops for generating the same result if we use materialized views: 21 loops -> 7 loops -> 4 loops -> 4 loops
- Query against the raw database requested the most effort. Using our pre-aggregated sales table already reduced the query time.
- Using `View2`, the *Grouping Compatibility Check* is also positive, because all attribute in our `GROUP BY` already used in `View2`'s `GROUP BY`.
- We already calculated the aggregation in `View2`. We can see when we use `View2` instead of `View1`, the execution time drastically reduced, the engine spares the aggregation time. The query optimizer determined *Aggregate Computability*.
- Not only the execution time is reduced, savings in planning time are also noticeable: 0.36ms vs 0.087ms

b)

Use SQL to materialize the following view:

```
CREATE MATERIALIZED VIEW View3 AS
   SELECT District, TimeId, DayOfWeek, ISBN, SUM(Amnt)
   FROM Sales NATURAL JOIN Customer NATURAL JOIN Time_Dim
   GROUP BY District, TimeId, DayOfWeek, ISBN;
```

Use PostgreSQL `EXPLAIN ANLYZE` (and `VACUUM ANALYZE`) command to get time needed to retrieve the `country` whose inhabitants spent the largest amount of money buying books when the SQL statement is issued against: 1. The "Book Orders Database", 2. The Data Mart, 3. The view View2, and 4. The view View3.

The Book Orders Database

```
EXPLAIN ANALYZE
SELECT
  country   AS Country,
  sum(amnt) AS Spending
FROM customer NATURAL JOIN
  (SELECT
     customer.customerid                                AS CustomerId,
     time.timeid                                        AS TimeId,
     book.isbn                                          AS ISBN,
     sum(order_detail.quantity * book.price) :: NUMERIC(6, 2)
      AS Amnt
   FROM
     book NATURAL JOIN order_detail NATURAL JOIN cust_order
      NATURAL JOIN customer NATURAL JOIN time
   GROUP BY customer.customerid, time.timeid, book.isbn
   ORDER BY CustomerId, TimeId, ISBN) AS sales
GROUP BY Country ORDER BY Spending DESC LIMIT 1;
                                                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=202.02..202.02 rows=1 width=48) (actual time=2.410..2.410 rows=1 loops
   ->  Sort  (cost=202.02..202.04 rows=7 width=48) (actual time=2.410..2.410 rows=1
         Sort Key: (sum(((sum(((order_detail.quantity)::numeric * book.price)))::num
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=201.90..201.99 rows=7 width=48) (actual time=2.395
               Group Key: customer.country
               ->  Hash Join  (cost=141.29..198.65 rows=649 width=30) (actual time=1
                     Hash Cond: (customer_1.customerid = customer.customerid)
                     ->  GroupAggregate  (cost=132.63..168.38 rows=1100 width=26) (c
                           Group Key: customer_1.customerid, "time".timeid, book.isb
                           ->  Sort  (cost=132.63..135.38 rows=1100 width=19) (actua
                                 Sort Key: customer_1.customerid, "time".timeid, boo
                                 Sort Method: quicksort  Memory: 134kB
                                 ->  Hash Join  (cost=27.82..77.06 rows=1100 width=1
                                       Hash Cond: (order_detail.isbn = book.isbn)
```

```
                                    ->  Hash Join  (cost=26.55..60.67 rows=1100 w
                                          Hash Cond: (order_detail.orderid = cust
                                          ->  Seq Scan on order_detail  (cost=0.0
                                          ->  Hash  (cost=23.77..23.77 rows=222 w
                                                Buckets: 1024  Batches: 1  Memory
                                                ->  Hash Join  (cost=13.45..23.77
                                                      Hash Cond: (cust_order.orde
                                                      ->  Hash Join  (cost=8.65..
                                                            Hash Cond: (cust_orde
                                                            ->  Seq Scan on cust_
                                                            ->  Hash  (cost=7.18.
                                                                  Buckets: 1024
                                                                  ->  Seq Scan or
                                                      ->  Hash  (cost=3.24..3.24
                                                            Buckets: 1024  Batche
                                                            ->  Seq Scan on "time
                                    ->  Hash  (cost=1.12..1.12 rows=12 width=9) (
                                          Buckets: 1024  Batches: 1  Memory Usage
                                          ->  Seq Scan on book  (cost=0.00..1.12
                        ->  Hash  (cost=7.18..7.18 rows=118 width=20) (actual time=0.03
                              Buckets: 1024  Batches: 1  Memory Usage: 14kB
                              ->  Seq Scan on customer  (cost=0.00..7.18 rows=118 width
 Planning time: 0.457 ms
 Execution time: 2.476 ms
(39 rows)

VACUUM ANALYZE;
VACUUM
```

21 loops

The Data Mart

```
EXPLAIN ANALYZE
SELECT
  country   AS Country,
  sum(amnt) AS Spending
FROM customer NATURAL JOIN sales
GROUP BY Country ORDER BY Spending DESC LIMIT 1;
                                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Limit  (cost=46.54..46.54 rows=1 width=48) (actual time=0.677..0.677 rows=1 loops=1
   ->  Sort  (cost=46.54..46.56 rows=7 width=48) (actual time=0.677..0.677 rows=1 lo
         Sort Key: (sum(sales.amnt)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=46.42..46.50 rows=7 width=48) (actual time=0.669..
               Group Key: customer.country
               ->  Hash Join  (cost=8.65..41.07 rows=1070 width=21) (actual time=0.0
                     Hash Cond: (sales.customerid = customer.customerid)
                     ->  Seq Scan on sales  (cost=0.00..17.70 rows=1070 width=9) (ac
                     ->  Hash  (cost=7.18..7.18 rows=118 width=20) (actual time=0.04
                           Buckets: 1024  Batches: 1  Memory Usage: 14kB
                           ->  Seq Scan on customer  (cost=0.00..7.18 rows=118 width
 Planning time: 0.170 ms
 Execution time: 0.703 ms
(14 rows)

VACUUM ANALYZE;
VACUUM
```

7 loops

The View2

```
EXPLAIN ANALYZE
SELECT
  country   AS Country,
  sum(sum)  AS Spending
FROM View2 NATURAL JOIN customer
GROUP BY Country ORDER BY Spending DESC LIMIT 1;
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=14.10..14.11 rows=1 width=48) (actual time=0.276..0.276 rows=1 loops=1
   ->  Sort  (cost=14.10..14.11 rows=1 width=48) (actual time=0.275..0.275 rows=1 lo
         Sort Key: (sum(view2.sum)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  GroupAggregate  (cost=14.07..14.09 rows=1 width=48) (actual time=0.236.
               Group Key: customer.country
               ->  Sort  (cost=14.07..14.08 rows=1 width=21) (actual time=0.227..0.2
                     Sort Key: customer.country
                     Sort Method: quicksort  Memory: 35kB
                     ->  Hash Join  (cost=9.25..14.06 rows=1 width=21) (actual time=
                           Hash Cond: ((view2.customerid = customer.customerid) AND
                           ->  Seq Scan on view2  (cost=0.00..3.32 rows=132 width=51
                           ->  Hash  (cost=7.18..7.18 rows=118 width=62) (actual tim
                                 Buckets: 1024  Batches: 1  Memory Usage: 19kB
                                 ->  Seq Scan on customer  (cost=0.00..7.18 rows=118
 Planning time: 0.426 ms
 Execution time: 0.303 ms
(17 rows)

VACUUM ANALYZE;
VACUUM
```

8 loops

The View3

```
EXPLAIN ANALYZE
SELECT
  c.country  AS Country,
  sum(sum)   AS Spending
FROM View3 NATURAL JOIN
  (SELECT DISTINCT district, country FROM customer) as c
GROUP BY Country ORDER BY Spending DESC LIMIT 1;
                                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Limit  (cost=47.54..47.55 rows=1 width=64) (actual time=0.768..0.768 rows=1 loops=1
   ->  Sort  (cost=47.54..47.59 rows=17 width=64) (actual time=0.767..0.767 rows=1 l
         Sort Key: (sum(view3.sum)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=47.25..47.46 rows=17 width=64) (actual time=0.759.
               Group Key: c.country
               ->  Hash Join  (cost=8.32..42.22 rows=1006 width=21) (actual time=0.0
                     Hash Cond: (view3.district = c.district)
                     ->  Seq Scan on view3  (cost=0.00..20.06 rows=1006 width=21) (a
                     ->  Hash  (cost=8.11..8.11 rows=17 width=32) (actual time=0.064
                           Buckets: 1024  Batches: 1  Memory Usage: 10kB
                           ->  Subquery Scan on c  (cost=7.77..8.11 rows=17 width=32
                                 ->  HashAggregate  (cost=7.77..7.94 rows=17 width=3
                                       Group Key: customer.district, customer.countr
                                       ->  Seq Scan on customer  (cost=0.00..7.18 ro
 Planning time: 0.148 ms
 Execution time: 0.808 ms
(17 rows)
```

9 loops

Findings:

- In the first case we have the same massive and slow query what we had in Question 4a. Using our Data Mart sales table improved our query heavily. From 21 loops to 7 loops. `Country` was not part of `sales` but *Functional Dependency* helped to optimize the query using `sales.customerid = customer.customerid` and it helps because `country` is functionally dependent on `customerid`.
- However, optimization with using View2 and View3 was not so efficient in this section.
- In case of **View2**, the *Data Sufficiency Check* is not satisfied, because the `country` is not subset of `View2`, however we still enjoy the advantage of `Functional Dependency` because of `customerid` is part of `View2` which helps to find the connected `country`. Plus we enjoy the advantages of the satisfied `Aggregate Computability` check.
- In case of **View3** the `Join Compatibility Check` is not satisfied without a little help. For this reason we have to join an extra query to satisfy the query, based on our *data hierarchy*. After our extra `Select` we satisfy the `Group Compatibility Check` (View3 has group by district) and

`Join Compatibility Check` also. Additionally we satisfy the `Aggregate Computability` check, because we use the aggregated result from View3.

Question 5 - Queries with WINDOW Function

(30 marks)

To answer the following two questions you will need to apply WINDOW function onto your datamart. Read PostgreSQL manual to find out more how PostgreSQL supports the WINDOW function.

a) (15 marks) Business analysts want to contrast the sum of amounts spent by customers buying books in `April` and `May` `2017` with the average amount spent by all customers from a city. So, in your answer, customers, represented by their `customerId` 's and `first names` , need to be `grouped by cities` . Also, place the average after the sum of amount column.

First, I run our report without using `WINDOW` .

```
-- Sum of amounts spent by customers in April and May in 2017:

SELECT
  customerid AS CustomerId,
  f_name     AS FirstName,
  sum(amnt)  AS SumOfSalesByCustomer
FROM sales NATURAL JOIN customer NATURAL JOIN time
WHERE (Month IN ('April', 'May')) AND (Year = 2017)
GROUP BY CustomerId, FirstName;

 customerid |      firstname       | sumofsalesbycustomer
------------+----------------------+----------------------
        109 | Neel                 |              1440.00
        118 | Daniel               |              1465.00
        112 | Bilal                |              1120.00
        116 | Adrian               |               765.00
        101 | Benjamin             |              1245.00
        103 | Mansi                |              3080.00
        106 | Li                   |              2035.00
         94 | Shweta               |              3615.00
        113 | Tao                  |              2055.00
         95 | Priyanka             |              1440.00
        100 | Zoltan               |              2710.00
        108 | Aaron                |              1555.00
        111 | Kaszandra            |              1360.00
        115 | Christopher          |              2065.00
        104 | Mansour              |              1425.00
        105 | Jessie               |              1670.00
         99 | Nathan               |               785.00
```

```
          98 | Valerie              |               925.00
         110 | Ronni                |              1490.00
         107 | Xiaoxing             |              1550.00
         114 | Harman               |               395.00
         117 | Lei                  |              1625.00
         102 | Leila                |              1165.00
          96 | Jovan                |               775.00
          97 | Cameron              |              1475.00
(25 rows)

-- Avg transactions speding by city in April and May in 2017:

SELECT
  city       AS City,
  avg(amnt)  AS AvgOfSalesByCity
FROM sales NATURAL JOIN customer NATURAL JOIN time
WHERE (Month IN ('April', 'May')) AND (Year = 2017)
GROUP BY city;

      city       |   avgofsalesbycity
-----------------+----------------------
 Lower Hutt      | 155.5000000000000000
 Christchurch    | 104.8611111111111111
 Wuhan           | 156.7647058823529412
 Budapest        | 225.8333333333333333
 Porirua         |  99.8076923076923077
 Sydney          | 163.2692307692307692
 Beijing         | 115.7500000000000000
 Skopje          |  77.5000000000000000
 Upper Hutt      | 135.4545454545454545
 Auckland        | 197.5000000000000000
 Wellington      | 121.5789473684210526
 Masterton       | 732.5000000000000000
 Mumbai          | 113.1818181818181818
(13 rows)
```

Please note, the above average number is the average spending on a transaction. I think, calculating average based on the whole spending in a period is more realistic and useful, so I show that solution later.

With `WINDOW`, using average calculated by transactions:

```
SELECT DISTINCT * FROM
  (
    SELECT
      customerid                                    AS CustomerId,
      f_name                                        AS FirstName,
      city                                          AS City,
      sum(amnt) OVER ( PARTITION BY customerId )    AS SumOfSalesByCustomer,
      avg(amnt) OVER ( PARTITION BY city )          AS AvgOfSalesByCity
    FROM sales NATURAL JOIN customer NATURAL JOIN time
    WHERE (Month IN ('April', 'May')) AND (Year = 2017)
  ) AS SalesReport ORDER BY City;
```

customerid	firstname	city	sumofsalesbycu	avgofsalesbycity
94	Shweta	Auckland	3615.00	197.50000000000000
95	Priyanka	Auckland	1440.00	197.50000000000000
113	Tao	Auckland	2055.00	197.50000000000000
107	Xiaoxing	Beijing	1550.00	115.75000000000000
116	Adrian	Beijing	765.00	115.75000000000000
100	Zoltan	Budapest	2710.00	225.83333333333333
98	Valerie	Christchurch	925.00	104.86111111111111
99	Nathan	Christchurch	785.00	104.86111111111111
115	Christopher	Christchurch	2065.00	104.86111111111111
108	Aaron	Lower Hutt	1555.00	155.50000000000000
118	Daniel	Masterton	1465.00	732.50000000000000
101	Benjamin	Mumbai	1245.00	113.18181818181818
97	Cameron	Porirua	1475.00	99.80769230769230
112	Bilal	Porirua	1120.00	99.80769230769230
96	Jovan	Skopje	775.00	77.50000000000000
102	Leila	Sydney	1165.00	163.26923076923076
103	Mansi	Sydney	3080.00	163.26923076923076
110	Ronni	Upper Hutt	1490.00	135.45454545454545
104	Mansour	Wellington	1425.00	121.57894736842105
109	Neel	Wellington	1440.00	121.57894736842105
111	Kaszandra	Wellington	1360.00	121.57894736842105
114	Harman	Wellington	395.00	121.57894736842105
105	Jessie	Wuhan	1670.00	156.76470588235294
106	Li	Wuhan	2035.00	156.76470588235294
117	Lei	Wuhan	1625.00	156.76470588235294

(25 rows)

This is my other report, where we calculate average spending by customer for the given period for the whole city and not average spending by transactions. In this case we create a materialized view about customer spending and we use this materialized view to present our report.

```
CREATE MATERIALIZED VIEW customer_spending AS
  SELECT
    customerid AS CustomerId,
    f_name     AS FirstName,
    city       AS City,
    sum(amnt)  AS AmountOfSpending
  FROM sales NATURAL JOIN customer NATURAL JOIN time
  WHERE year = 2017 AND month IN ('April', 'May')
  GROUP BY CustomerId, FirstName, City
  ORDER BY City;
SELECT 25
```

In this case the average reflects the spending of a customer in the whole period in a city.

```
SELECT
  CustomerId,
  FirstName,
  City,
  AmountOfSpending,
  avg(AmountOfSpending) OVER CityWin AS AvgSpendingByCity
FROM customer_spending
WINDOW CityWin AS ( PARTITION BY city )
ORDER BY city;
```

```
 customerid |      firstname       |      city      | amountofsp|  avgspendingbycity
------------+----------------------+----------------+-----------+---------------------
         94 | Shweta               | Auckland       |   3615.00 | 2370.0000000000000000
         95 | Priyanka             | Auckland       |   1440.00 | 2370.0000000000000000
        113 | Tao                  | Auckland       |   2055.00 | 2370.0000000000000000
        107 | Xiaoxing             | Beijing        |   1550.00 | 1157.5000000000000000
        116 | Adrian               | Beijing        |    765.00 | 1157.5000000000000000
        100 | Zoltan               | Budapest       |   2710.00 | 2710.0000000000000000
         98 | Valerie              | Christchurch   |    925.00 | 1258.3333333333333333
         99 | Nathan               | Christchurch   |    785.00 | 1258.3333333333333333
        115 | Christopher          | Christchurch   |   2065.00 | 1258.3333333333333333
        108 | Aaron                | Lower Hutt     |   1555.00 | 1555.0000000000000000
        118 | Daniel               | Masterton      |   1465.00 | 1465.0000000000000000
        101 | Benjamin             | Mumbai         |   1245.00 | 1245.0000000000000000
         97 | Cameron              | Porirua        |   1475.00 | 1297.5000000000000000
        112 | Bilal                | Porirua        |   1120.00 | 1297.5000000000000000
         96 | Jovan                | Skopje         |    775.00 |  775.0000000000000000
        102 | Leila                | Sydney         |   1165.00 | 2122.5000000000000000
        103 | Mansi                | Sydney         |   3080.00 | 2122.5000000000000000
        110 | Ronni                | Upper Hutt     |   1490.00 | 1490.0000000000000000
        104 | Mansour              | Wellington     |   1425.00 | 1155.0000000000000000
        109 | Neel                 | Wellington     |   1440.00 | 1155.0000000000000000
        111 | Kaszandra            | Wellington     |   1360.00 | 1155.0000000000000000
        114 | Harman               | Wellington     |    395.00 | 1155.0000000000000000
        105 | Jessie               | Wuhan          |   1670.00 | 1776.6666666666666667
        106 | Li                   | Wuhan          |   2035.00 | 1776.6666666666666667
        117 | Lei                  | Wuhan          |   1625.00 | 1776.6666666666666667
(25 rows)
```

b) (15 marks) Business analysts want to contrast the daily sums of amounts spent by all customers from a city buying books in `April` and `May` `2017` with the cumulative sum from the start of `April` including the current day. In the query result, you need to display the following columns in the following order: `city`, `timeid`, `day`, `sum(amnt)`, `cumulative_sum`.

I found more elegant if I use a materialized view for aggregating daily spending by city and by day. In a second query, using WINDOW, we can generate the cumulative aggregate.

```
CREATE MATERIALIZED VIEW sum_per_day_per_city AS
  SELECT
    city,
    timeid,
    OrderDate AS day,
    sum(amnt) AS SumSpending
  FROM sales NATURAL JOIN time NATURAL JOIN customer
  WHERE Month IN ('April', 'May') AND Year = 2017
  GROUP BY city, timeid, OrderDate
  ORDER BY city, timeid;

SELECT
  city,
  timeid,
  day,
  SumSpending                    AS "sum(amnt)",
  sum(SumSpending) OVER WinCity AS cumulative_sum
FROM sum_per_day_per_city
WINDOW WinCity AS ( PARTITION BY city ORDER BY timeid )
ORDER BY city, timeid;
```

city	timeid	day	sum(amnt)	cumulative_sum
Auckland	119	2017-04-23	360.00	360.00
Auckland	120	2017-04-29	2250.00	2610.00
Auckland	121	2017-04-30	2805.00	5415.00
Auckland	122	2017-05-05	1695.00	7110.00
Beijing	117	2017-04-21	1550.00	1550.00
Beijing	124	2017-05-15	765.00	2315.00
Budapest	111	2017-04-15	2710.00	2710.00
Christchurch	110	2017-04-14	925.00	925.00
Christchurch	111	2017-04-15	785.00	1710.00
Christchurch	122	2017-05-05	175.00	1885.00
Christchurch	123	2017-05-06	1810.00	3695.00
Christchurch	124	2017-05-15	80.00	3775.00
Lower Hutt	118	2017-04-22	1555.00	1555.00
Masterton	124	2017-05-15	1465.00	1465.00
Mumbai	112	2017-04-16	1245.00	1245.00
Porirua	108	2017-04-12	170.00	170.00
Porirua	109	2017-04-13	1135.00	1305.00
Porirua	110	2017-04-14	170.00	1475.00
Porirua	122	2017-05-05	1120.00	2595.00
Skopje	108	2017-04-12	775.00	775.00
Sydney	112	2017-04-16	850.00	850.00
Sydney	113	2017-04-17	1305.00	2155.00
Sydney	114	2017-04-18	2090.00	4245.00
Upper Hutt	118	2017-04-22	1490.00	1490.00
Wellington	114	2017-04-18	1425.00	1425.00

```
 Wellington      |     118 | 2017-04-22 |   1440.00 |         2865.00
 Wellington      |     119 | 2017-04-23 |   1360.00 |         4225.00
 Wellington      |     123 | 2017-05-06 |    395.00 |         4620.00
 Wuhan           |     115 | 2017-04-19 |   1735.00 |         1735.00
 Wuhan           |     116 | 2017-04-20 |   1525.00 |         3260.00
 Wuhan           |     117 | 2017-04-21 |    195.00 |         3455.00
 Wuhan           |     118 | 2017-04-22 |    250.00 |         3705.00
 Wuhan           |     124 | 2017-05-15 |   1625.00 |         5330.00
(33 rows)
```

The following implementation using a nested query, but I think it is not easy to maintain for long term. I would prefer the above, cleaner and simple implementation in real world scenario.

```
SELECT
  city,
  timeid,
  OrderDate                     AS day,
  SumSpending                   AS "sum(amnt)",
  sum(SumSpending) OVER WinCity AS cumulative_sum
FROM (
       SELECT DISTINCT
         city,
         timeid,
         orderdate,
         sum(amnt) OVER WinDate AS SumSpending
       FROM sales NATURAL JOIN customer NATURAL JOIN time
       WHERE Month IN ('April', 'May') AND Year = 2017
       WINDOW WinDate AS ( PARTITION BY city, timeid )
     ) AS sum_per_day_per_city
WINDOW WinCity AS ( PARTITION BY city ORDER BY timeid )
ORDER BY city, timeid;
      city       | timeid |    day     | sum(amnt) | cumulative_sum
-----------------+--------+------------+-----------+----------------
 Auckland        |     119 | 2017-04-23 |    360.00 |         360.00
 Auckland        |     120 | 2017-04-29 |   2250.00 |        2610.00
 Auckland        |     121 | 2017-04-30 |   2805.00 |        5415.00
 Auckland        |     122 | 2017-05-05 |   1695.00 |        7110.00
 Beijing         |     117 | 2017-04-21 |   1550.00 |        1550.00
 Beijing         |     124 | 2017-05-15 |    765.00 |        2315.00
 Budapest        |     111 | 2017-04-15 |   2710.00 |        2710.00
 Christchurch    |     110 | 2017-04-14 |    925.00 |         925.00
 Christchurch    |     111 | 2017-04-15 |    785.00 |        1710.00
 Christchurch    |     122 | 2017-05-05 |    175.00 |        1885.00
 Christchurch    |     123 | 2017-05-06 |   1810.00 |        3695.00
 Christchurch    |     124 | 2017-05-15 |     80.00 |        3775.00
 Lower Hutt      |     118 | 2017-04-22 |   1555.00 |        1555.00
 Masterton       |     124 | 2017-05-15 |   1465.00 |        1465.00
```

```
 Mumbai          |     112 | 2017-04-16 |   1245.00 |         1245.00
 Porirua         |     108 | 2017-04-12 |    170.00 |          170.00
 Porirua         |     109 | 2017-04-13 |   1135.00 |         1305.00
 Porirua         |     110 | 2017-04-14 |    170.00 |         1475.00
 Porirua         |     122 | 2017-05-05 |   1120.00 |         2595.00
 Skopje          |     108 | 2017-04-12 |    775.00 |          775.00
 Sydney          |     112 | 2017-04-16 |    850.00 |          850.00
 Sydney          |     113 | 2017-04-17 |   1305.00 |         2155.00
 Sydney          |     114 | 2017-04-18 |   2090.00 |         4245.00
 Upper Hutt      |     118 | 2017-04-22 |   1490.00 |         1490.00
 Wellington      |     114 | 2017-04-18 |   1425.00 |         1425.00
 Wellington      |     118 | 2017-04-22 |   1440.00 |         2865.00
 Wellington      |     119 | 2017-04-23 |   1360.00 |         4225.00
 Wellington      |     123 | 2017-05-06 |    395.00 |         4620.00
 Wuhan           |     115 | 2017-04-19 |   1735.00 |         1735.00
 Wuhan           |     116 | 2017-04-20 |   1525.00 |         3260.00
 Wuhan           |     117 | 2017-04-21 |    195.00 |         3455.00
 Wuhan           |     118 | 2017-04-22 |    250.00 |         3705.00
 Wuhan           |     124 | 2017-05-15 |   1625.00 |         5330.00
(33 rows)
```

Comparing the solutions 1 materialized view with 1 extra query VS 1 massive query:

Materialized View:

```
EXPLAIN ANALYZE
CREATE MATERIALIZED VIEW sum_per_day_per_city AS
  SELECT
    city,
    timeid,
    OrderDate AS day,
    sum(amnt) AS SumSpending
  FROM sales NATURAL JOIN time NATURAL JOIN customer
  WHERE Month IN ('April', 'May') AND Year = 2017
  GROUP BY city, timeid, OrderDate
  ORDER BY city, timeid;

QUERY PLAN
-------------------------------------------------------------------------------------
 GroupAggregate  (cost=35.83..36.90 rows=43 width=56) (actual time=1.756..1.963 rows
   Group Key: customer.city, sales.timeid, "time".orderdate
   ->  Sort  (cost=35.83..35.94 rows=43 width=29) (actual time=1.745..1.816 rows=272
         Sort Key: customer.city, sales.timeid, "time".orderdate
         Sort Method: quicksort  Memory: 46kB
         ->  Nested Loop  (cost=4.06..34.66 rows=43 width=29) (actual time=0.492..1.
               ->  Hash Join  (cost=3.92..26.07 rows=43 width=17) (actual time=0.486
                     Hash Cond: (sales.timeid = "time".timeid)
                     ->  Seq Scan on sales  (cost=0.00..17.70 rows=1070 width=13) (a
                     ->  Hash  (cost=3.86..3.86 rows=5 width=8) (actual time=0.039..
                           Buckets: 1024  Batches: 1  Memory Usage: 9kB
                           ->  Seq Scan on "time"  (cost=0.00..3.86 rows=5 width=8)
                                 Filter: ((month = ANY ('{April,May}'::bpchar[])) AN
                                 Rows Removed by Filter: 107
               ->  Index Scan using customer_pkey on customer  (cost=0.14..0.19 rows
                     Index Cond: (customerid = sales.customerid)
 Planning time: 0.267 ms
 Execution time: 4.403 ms
(18 rows)
```

7 loops + 272 loops = 279 loops

One small query:

```
EXPLAIN ANALYZE
SELECT
  city,
  timeid,
  day,
  SumSpending                    AS "sum(amnt)",
  sum(SumSpending) OVER WinCity AS cumulative_sum
FROM sum_per_day_per_city
WINDOW WinCity AS ( PARTITION BY city ORDER BY timeid )
ORDER BY city, timeid;

QUERY PLAN
------------------------------------------------------------------------------------
 WindowAgg  (cost=2.16..2.82 rows=33 width=61) (actual time=0.036..0.064 rows=33 loo
   ->  Sort  (cost=2.16..2.24 rows=33 width=29) (actual time=0.028..0.030 rows=33 lo
         Sort Key: city, timeid
         Sort Method: quicksort  Memory: 27kB
         ->  Seq Scan on sum_per_day_per_city  (cost=0.00..1.33 rows=33 width=29) (a
 Planning time: 0.061 ms
 Execution time: 0.088 ms
(7 rows)
```

3 loops

Our massive query:

```
EXPLAIN ANALYZE
SELECT
  city,
  timeid,
  OrderDate                     AS day,
  SumSpending                   AS "sum(amnt)",
  sum(SumSpending) OVER WinCity AS cumulative_sum
FROM (
       SELECT DISTINCT
         city,
         timeid,
         orderdate,
         sum(amnt) OVER WinDate AS SumSpending
       FROM sales NATURAL JOIN customer NATURAL JOIN time
       WHERE Month IN ('April', 'May') AND Year = 2017
       WINDOW WinDate AS ( PARTITION BY city, timeid )
     ) AS sum_per_day_per_city
WINDOW WinCity AS ( PARTITION BY city ORDER BY timeid )
ORDER BY city, timeid;
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 WindowAgg  (cost=37.85..39.57 rows=43 width=88) (actual time=1.113..1.205 rows=33 l
   ->  Unique  (cost=37.85..38.39 rows=43 width=56) (actual time=1.110..1.181 rows=3
         ->  Sort  (cost=37.85..37.96 rows=43 width=56) (actual time=1.109..1.125 ro
               Sort Key: customer.city, sales.timeid, "time".orderdate, (sum(sales.a
               Sort Method: quicksort  Memory: 46kB
               ->  WindowAgg  (cost=35.83..36.69 rows=43 width=56) (actual time=0.86
                     ->  Sort  (cost=35.83..35.94 rows=43 width=29) (actual time=0.8
                           Sort Key: customer.city, sales.timeid
                           Sort Method: quicksort  Memory: 46kB
                           ->  Nested Loop  (cost=4.06..34.66 rows=43 width=29) (act
                                 ->  Hash Join  (cost=3.92..26.07 rows=43 width=17)
                                       Hash Cond: (sales.timeid = "time".timeid)
                                       ->  Seq Scan on sales  (cost=0.00..17.70 rows
                                       ->  Hash  (cost=3.86..3.86 rows=5 width=8) (a
                                             Buckets: 1024  Batches: 1  Memory Usage
                                             ->  Seq Scan on "time"  (cost=0.00..3.8
                                                   Filter: ((month = ANY ('{April,Ma
                                                   Rows Removed by Filter: 107
                                 ->  Index Scan using customer_pkey on customer  (co
                                       Index Cond: (customerid = sales.customerid)
 Planning time: 0.329 ms
 Execution time: 1.245 ms
(22 rows)
```

10 loops + 272 loops = 282 loops

Interesting to see, that both solution using the same amount of loop (282). In the first case creating the materialized view needs more time, but our little query much much faster.

I think separating, splitting up queries help us to write cleaner more maintainable code and can speed up our operation, especially if we optimize our materialized views properly.