Data Management Systems- Sales Cube

1. Data entry

After importing data to CSV tables, I prepared it based on my schema and made proper CSV files. For example, I have used the following code in Python for populating the States table.

```
import psycopg2
import csv
import os

# Connect to PostgreSQL database
conn = psycopg2.connect(
    host="localhost",
    database="Sales_Cube",
    user="postgres",
    password="************"
)
cursor = conn.cursor()

# Define the file paths relative to the script's location
states_file = '/Users/zeinab/Documents/DSE/SQL/HW/Sales/states_sales.csv'

with open(states_file, 'r') as f:
    cursor.copy_expert(f"COPY States FROM STDIN WITH CSV HEADER;", file=f)

# Commit changes and close the connection
conn.commit()
conn.close()
```

2. Performance Optimization

Data size

States: 50 rows, state_id, state_name
Customers: 887970 rows, customer_id, customer_name, state_id
Products: 100000 rows, product_id, product_name, product_list_price
Categories: 100 rows, category_id, category_name, category_description
ProductCategories: 100000 rows, productcategory_id, category_id, product_id
Sales: 4218264 rows, sales_id, customer_id, product_id, product_quantity, discount

1. The total sales for each customer

This query uses all the customer_ids from Customer table (887970 rows) and calculates the product quantity and total dollar value for each customer. We have 4218264 rows of data in the Sales table and 100000 products. At last, it is supposed to return tuples of customers and total sales for each of them. Based on the question's request to return all the customers, even the ones with no sales record, the number of output tuples is equal to the number of customers, which is 887970.

```
SELECT cu.customer_id,
COALESCE(SUM(s.product_quantity), 0) as total_quantity_sold,
COALESCE(SUM(product_list_price*(1-s.discount)),0) as total_value
FROM Customers cu
LEFT JOIN Sales s
ON cu.customer_id = s.customer_id
LEFT JOIN Products pr
ON pr.product_id = s.product_id
GROUP BY cu.customer_id
ORDER BY total_value DESC;
```

Adding indexes

For this query, we are retrieving aggregated information about customers, their total quantity sold, and the corresponding total value. In this case, the suggested indexes for the first experiment would be as follows:

1. Index on Customers table: customer_id in the Customers table, as it is the primary key used for joining.
2. Index on Sales table: two indexes on customer_id and product_id in the Sales table, as these columns are involved in the join conditions.
3. Index on Products table: Index on product_id in the Products table to optimize the join with the Sales table.

The suggested indexes aim to enhance the efficiency of the JOIN operations and GROUP BY clause in the query.

Experiment 1

```
-- Index on Customers table
CREATE INDEX idx_customer_customer_id ON Customers(customer_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Sales table
```

```
CREATE INDEX idx_sales_product_id ON Sales(product_id);
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Experiment 2

```
-- Index on Customers table
CREATE INDEX idx_customer_customer_id ON Customers(customer_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Sales table
CREATE INDEX idx_sales_cuspro_id ON Sales(customer_id, product_id);
```

Summary

1- Total sales for each customer			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:05.312	cost=715614.59..717834.52	-
1	00:00:05.642	cost=715614.59..717834.52	Customer_customer_id Product_product_id sales_customer_id sales_product_id
	00:00:05.569	cost=715614.59..717834.52	Customer_customer_id Product_product_id sales_cuspro_id

No indexes have been used according to the EXPLAIN statement in the both experiments. In this query, the database optimizer may choose not to utilize indexes even when they are available.

2. Total sales for each state

This query, similar to the last one, uses all the state_ids from the States table (50 rows) and calculates the product quantity and total dollar value for each customer. We have 4218264 rows of data in the Sales table and 100000 products. In this query we want to return total sales for each state. Considering all the states regardless of their sales (even zero), the outcome of this query is equal to the number of states, 50.

```
SELECT st.state_name,
COALESCE(SUM(s.product_quantity), 0) as total_quantity_sold,
COALESCE(SUM(product_list_price*(1-s.discount)),0) as total_value
FROM States st
```

```
LEFT JOIN Customers cu
ON st.state_id = cu.state_id
LEFT JOIN Sales s
ON cu.customer_id = s.customer_id
LEFT JOIN Products pr
ON pr.product_id = s.product_id
GROUP BY st.state_name
ORDER BY total_value DESC;
```

Adding indexes

To optimize the performance of this query, these indexes may enhance efficiency. Here are suggested indexes for the involved tables:

1. Index on States table: Index on state_id in the States table, as it is used in the join condition.
2. Index on Customers table: Index on customer_id and state_id in the Customers table to optimize the join with the States table.
3. Index on Sales table: index on customer_id and product_id in the Sales table, as these columns are involved in the join conditions.
4. Index on Products table: Index on product_id in the Products table to optimize the join with the Sales table.

Experiment 1

```
-- Index on Customers table
CREATE INDEX idx_customer_customer_id ON Customers(customer_id);
CREATE INDEX idx_customer_state_id ON Customers(state_id);

-- Index on States table
CREATE INDEX idx_state_state_id ON States(state_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Sales table
CREATE INDEX idx_sales_product_id ON Sales(product_id);
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Experiment 2

```
-- Index on Customers table
CREATE INDEX idx_customer_cussta ON Customers(customer_id, state_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);
```

```
-- Index on Sales table
CREATE INDEX idx_sales_cuspro_id ON Sales(customer_id, product_id);
```

Summary

2- Total sales for each state			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:03.941	cost=242634.34..242634.46	-
1	00:00:03.635	cost=242634.34..242634.46	customer_ customer_id, customer_state_id State_state_id Product_product_id sales_ customer_id sales_product_id
2	00:00:03.658	cost=242634.34..242634.46	customer_cussta State_state_id Product_product_id sales_cuspro_id

In this query, like the previous one, the absence of index utilization persisted. The cost remained unaltered, while a marginal reduction in execution time was observed. Surprisingly, a parallel outcome ensued during the subsequent attempt. Even the composite indexes failed to yield the anticipated improvements in performance.

3. Show the total sales for each product for a given customer

For this question, we want to calculate the total sales for each product for each customer. We have 4218264 rows of data in the Sales table and 887970 customers. So, our output is (product_id, custoemr_id, total sales). According to the question, our data is in the Sales table because we need only sold products to each customer.

```
SELECT s.product_id, s.customer_id,
          COALESCE(SUM(product_list_price*(1-s.discount)),0) as total
    FROM Sales s
    LEFT JOIN Products pr
    ON pr.product_id = s.product_id
    GROUP BY s.customer_id, s.product_id
    ORDER BY s.customer_id, total DESC;
```

Adding indexes

To optimize the performance of this query, these indexes may enhance efficiency. Here are suggested indexes for the involved tables:

1. Index on Customers table: Index on customer_id and state_id in the Customers table to optimize the join with the States table.
2. Index on Sales table: index on customer_id and product_id in the Sales table, as these columns are involved in the join conditions.
3. Index on Products table: Index on product_id in the Products table to optimize the join with the Sales table.

Experiment 1

```
-- Index on Customers table
CREATE INDEX idx_customer_customer_id ON Customers(customer_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Sales table
CREATE INDEX idx_sales_product_id ON Sales(product_id);
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Summary

3- Total sales for product state by each customer			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:06.888	cost=295233.41..475431.11	-
1	00:00:06.450	cost=295233.41..475431.11	customer_ customer_id, Product_product_id sales_ customer_id sales_product_id
2	00:00:06.657	cost=295233.41..475431.11	customer_ customer_id, Product_product_id sales_cuspro_id

This query, similar to the previous ones, appears to have refrained from utilizing indexes. Despite conducting two separate experiments involving both single and composite indexes, the cost of execution maintained its unaltered state.

4. The total sales for each product and customer

For returning all the tuples of customers and products, even if a customer did not buy that product, we need to consider all the possible selection of customers and products. Therefore, the volume of the output data would be around 89 billion rows (887970 * 100000). I have tried to run this code many times but there is not enough memory on my system. So, I reduced the number of customers in a modified version and ran it for all the products (output: 50’000’000 tuples).

```
SELECT
  cu.customer_id,
  pr.product_id,
 COALESCE(SUM(product_list_price*(1-s.discount)),0) as total
FROM
  Products pr
   CROSS JOIN Customers cu
  LEFT JOIN Sales s
    ON pr.product_id = s.product_id
             AND cu.customer_id = s.customer_id
GROUP BY pr.product_id, cu.customer_id
ORDER BY cu.customer_id, total DESC;
```

Modified version

```
SELECT
  cu.customer_id,
  pr.product_id,
 COALESCE(SUM(product_list_price*(1-s.discount)),0) as total
FROM
  Products pr
   CROSS JOIN Customers cu
  LEFT JOIN Sales s
    ON pr.product_id = s.product_id
             AND cu.customer_id = s.customer_id
WHERE cu.customer_id < 500
GROUP BY pr.product_id, cu.customer_id
ORDER BY cu.customer_id, total DESC;
```

This query needs a cross join to be completed. In the cross joins, indexes cannot be helpful. In this case, I had to add a WHERE clause to prevent crashing the system, so an index on that (customer_id) may reduce the time of run.

Experiment 1

```
-- Index on Sales table
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Summary

4- The total sales for each product and customer			
Experiment	Time	Cost	Indexes
0 (no indices)	00:01:08.901	cost=21222972.83..21345972.83	-
1		cost=21222972.83..21345972.83	customer_ customer_id

5. Total sales for each product category and state

In this question we should find the total sales for each product category and state. I have 100 categories and 50 states. Hence, the result would be 5000 tuples of category_id, state_id and total sales for them.

```
SELECT
   pc.category_id,
   st.state_name,
   COALESCE(SUM(product_list_price*(1-s.discount)),0) AS total
FROM
  Sales s
  LEFT JOIN Products pr
    ON pr.product_id = s.product_id
  JOIN Product_Category pc
    ON pr.product_id = pc.product_id
  JOIN Customers cu
    ON cu.customer_id = s.customer_id
  JOIN States st
    ON cu.state_id = st.state_id
GROUP BY pc.category_id, st.state_name
ORDER BY total DESC;
```

Adding indexes

To optimize the performance of this query, these indexes may enhance efficiency. Here are suggested indexes for the involved tables:

1. Index on States table: Index on state_id in the States table, as it is used in the join condition.
2. Index on Customers table: Index on customer_id and state_id in the Customers table to optimize the join with the States table.

3. Index on Sales table: index on customer_id and product_id in the Sales table, as these columns are involved in the join conditions.
4. Index on Products table: Index on product_id in the Products table to optimize the join with the Sales table.
5. Index on Product_Category table: index on category_id in this table, as the column is involved in the group by and join conditions.

Experiment 1

```
-- Index on Customers table
CREATE INDEX idx_customer_customer_id ON Customers(customer_id);
CREATE INDEX idx_customer_state_id ON Customers(state_id);

-- Index on States table
CREATE INDEX idx_state_state_id ON States(state_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Product_Category table
CREATE INDEX idx_pc_product_id ON Product_Category(product_id);
CREATE INDEX idx_pc_category_id ON Product_Category(category_id);

-- Index on Sales table
CREATE INDEX idx_sales_product_id ON Sales(product_id);
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Summary

5- Total sales for each product category and state			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:01.656	cost=145495.74..145508.24	-
1	00:00:01.750	cost=145495.74..145508.24	customer_customer_id customer_state_id state_state_id product_product_id pc_category_id sales_product_id sales_customer_id
2	00:00:01.588	cost=145495.74..145508.24	Customer_cussta state_state_id product_product_id

			Pc_category_id sales_cuspro_id

This query, like its predecessors, seems not to have taken advantage of indexes. Despite applying two distinct experiments, one with single indexes and another with composite indexes, the execution cost remained consistently unchanged.

6. Total sales for each one of the top 20 product categories and top 20 customers

In this query the result should be (top 20 category_id, top 20 customer_id, quantity sold and dollar value). So, we are going to have 400 rows of results.

```
WITH
-- top 20 product categories
TopCategory AS (
SELECT
            pc.category_id,
    COALESCE(SUM(product_list_price*(1-s.discount)),0) AS total_TopCategory
FROM
    Sales s
    LEFT JOIN Products pr
      ON pr.product_id = s.product_id
    JOIN Product_Category pc
      ON pr.product_id = pc.product_id
GROUP BY pc.category_id
ORDER BY total_TopCategory DESC
LIMIT 20),
-- rank by category
rank_category AS (
            SELECT
        ROW_NUMBER() OVER (ORDER BY total_TopCategory DESC) AS rank_category,
        category_id,
        total_TopCategory
    FROM
                 TopCategory),
-- top 20 customers
TopCustomers AS (
  SELECT
              cu.customer_id,
              COALESCE(SUM(product_list_price*(1-s.discount)),0) AS total_TopCustomers
  FROM
              Customers cu
              LEFT JOIN Sales s
                ON cu.customer_id = s.customer_id
              LEFT JOIN Products pr
                ON pr.product_id = s.product_id
  GROUP BY cu.customer_id
  ORDER BY total_TopCustomers DESC
  LIMIT 20),
-- rank by customer
rank_customer AS (
              SELECT
         ROW_NUMBER() OVER (ORDER BY total_TopCustomers DESC) AS rank_customer,
         customer_id,
         total_TopCustomers
    FROM
                   TopCustomers),
-- category-aware sales
ProductSales AS (
  SELECT
    s.customer_id,
    s.product_id,
    pc.category_id,
    s.product_quantity,
    s.discount,
    pr.product_list_price
  FROM
    Sales s
    JOIN Products pr
                ON s.product_id = pr.product_id
              JOIN Product_Category pc
      ON pr.product_id = pc.product_id)

SELECT
  rank_category.category_id,
  rank_category.rank_category,
  rank_customer.customer_id,
  rank_customer.rank_customer,
  COALESCE(SUM(s.product_quantity),0) as quantity,
  COALESCE(SUM(s.product_list_price*(1-s.discount)),0) as dollar_value
FROM
  rank_customer
  CROSS JOIN rank_category
  LEFT JOIN ProductSales s
              ON rank_customer.customer_id = s.customer_id
              AND rank_category.category_id = s.category_id
GROUP BY rank_category.category_id, rank_category.rank_category,
         rank_customer.customer_id, rank_customer.rank_customer
ORDER BY rank_customer.rank_customer, rank_category.rank_category;
```

Adding indexes

To optimize the performance of this query, these indexes may enhance efficiency. Here are suggested indexes for the involved tables:

1. Index on States table: Index on state_id in the States table, as it is used in the join condition.
2. Index on Customers table: Index on customer_id and state_id in the Customers table to optimize the join with the States table.
3. Index on Sales table: index on customer_id and product_id in the Sales table, as these columns are involved in the join conditions.
4. Index on Products table: Index on product_id in the Products table to optimize the join with the Sales table.
5. Index on Product_Category table: index on pc_product_id in this table, as the column is involved in the group by and join conditions.

Experiment 1

```
-- Index on Customers table
CREATE INDEX idx_customer_customer_id ON Customers(customer_id);
CREATE INDEX idx_customer_state_id ON Customers(state_id);

-- Index on States table
CREATE INDEX idx_state_state_id ON States(state_id);

-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Product_Category table
CREATE INDEX idx_pc_product_id ON Product_Category(product_id);

-- Index on Sales table
CREATE INDEX idx_sales_product_id ON Sales(product_id);
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Experiment 2

```
-- Index on Products table
CREATE INDEX idx_product_product_id ON Products(product_id);

-- Index on Product_Category table
CREATE INDEX idx_pc_product_id ON Product_Category(product_id);

-- Index on Sales table
CREATE INDEX idx_sales_customer_id ON Sales(customer_id);
```

Summary

6- Total sales and quantity for top 20 category and top 20 customers			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:08.096	ost=884030.88..884044.8	-
1	00:00:05.766	cost=649009.44..649023.44	customer_customer_id product_product_id Pc_product_id sales_product_id sales_customer_id
2	00:00:05.421	cost=649009.44..649023.44	product_product_id Pc_product_id Sales_customer_id

In the initial attempt, I applied indexes to the Customers, Products, Product_Category, and Sales tables. According to the EXPLAIN statement, the indexes on product_product_id, Pc_product_id, and Sales_customer_id were utilized, resulting in a 26% improvement in cost. Subsequently, for the next experiment, I omitted the indexes that were not used. Despite dropping the unused indexes, the cost remained constant, with only a slight reduction in execution time.

However, a noteworthy observation emerged. The query selectively utilized indexes only when an index was present on pc_product_id. Intriguingly, upon dropping this specific index and attempting others, such as product_product_id, the query refrained from utilizing them, and the cost remained unaltered.

Note

About the question 1 to 5, the database optimizer may choose not to utilize indexes even when they are available, and based on my research, this behavior can be influenced by various factors:

- If the number of distinct values or percentage of rows selected of the indexed columns is not substantial, the optimizer might decide that a full table scan is more efficient than using the index.
- For smaller tables, a full table scan may be more efficient than navigating the index structure, especially if a significant portion of the table needs to be retrieved.
- The database optimizer uses a cost-based approach to choose the most efficient execution plan. If it determines that the cost of using an index is comparable to or higher than a full table scan, it may opt for the latter.