Database Lab Report 6

Course: Database Management
Lab Number: *Lab #6*
Date: *2025-02-25*
Name: *Ryan Munger*

1. Objective

Write some fun yet difficult SQL queries. Now you've got to earn the points

2. Lab Setup

A fresh CAP database and a can-do attitude

3. Procedure

Part 1: *Queries*

For each question, write the query. Then, use AI to generate the queries and grade its response.

1. Display the cities that makes the most different kinds of products. Experiment with the rank() function

```
-- Me: Intuition
SELECT city
FROM Products
GROUP BY city
HAVING COUNT(*) = (
                    SELECT MAX(count)
                    FROM (
                          SELECT COUNT(*) AS count
                          FROM Products
                          GROUP BY city
                         )
                  );
-- Me: Rank()
SELECT city, counts
FROM (
      SELECT city, COUNT(*) AS counts, RANK() OVER (ORDER BY COUNT(*) DESC) AS
rank
      FROM Products
      GROUP BY city
     );
-- WHERE rank = 1;
-- AI
SELECT city, COUNT(DISTINCT prodId), RANK() OVER (ORDER BY COUNT(DISTINCT prodId)
DESC) AS rank
FROM Products
GROUP BY city;
```

Grade: 9.5/10; DISTINCT prodId - PK, must be distinct already. Some indentation would have been nice as well.

2. Display the names of products whose priceUSD is less than 1% of the average priceUSD, in alphabetical order. from A to Z

```
-- Me
SELECT name
FROM Products
WHERE priceUSD < (
                    SELECT AVG(priceUSD) * 0.01
                    FROM Products
                 )
ORDER BY name ASC;
-- AI
SELECT name
FROM Products
WHERE priceUSD < (SELECT AVG(priceUSD) * 0.01 FROM Products)
ORDER BY name;
```

Grade: 9.9/10; Specify ASC!

3. Display the customer last name, product id ordered, and the totalUSD for all orders made in March of any year, sorted by totalUSD from low to high.

```
-- Me
SELECT p.lastName, o.prodId, o.totalUSD
FROM Orders o INNER JOIN People p ON p.pid = o.custId
WHERE extract(month from o.dateOrdered) = 3
ORDER BY o.totalUSD ASC;
-- AI
SELECT p.lastName, o.prodId, o.totalUSD
FROM Orders o
JOIN Customers c ON o.custId = c.pid
JOIN People p ON c.pid = p.pid
WHERE EXTRACT(MONTH FROM o.dateOrdered) = 3
ORDER BY o.totalUSD;
```

Grade: 10/10; We didn't really need to go through the Customers table here but it could be nice to be explicit.

4. Display the last name of all customers (in reverse alphabetical order) and their total ordered by customer, and nothing more. Use coalesce to avoid showing NULL totals.

```
-- Me
SELECT p.lastName, COALESCE(SUM(o.quantityOrdered), 0) AS totalOrdered
FROM People p INNER JOIN Customers c ON c.pid = p.pid
              LEFT OUTER JOIN Orders o ON o.custId = c.pid
```

```
GROUP BY p.lastName
ORDER BY p.lastName DESC;
-- AI
SELECT p.lastName, COALESCE(SUM(o.totalUSD), 0) AS total_ordered
FROM Customers c
JOIN People p ON c.pid = p.pid
LEFT JOIN Orders o ON c.pid = o.custId
GROUP BY p.lastName
ORDER BY p.lastName DESC;
```

Grade: 10/10. I took total ordered to be totalQuantityOrdered while AI took it to mean totalUSDOrdered.

5. Display the names of all customers who bought products from agents based in Regina along with the names of the products they ordered, and the names of the agents who sold it to them.

```
-- Me
SELECT pc.firstName || ' ' || pc.lastName AS custName, pa.firstName || ' ' ||
pa.lastName AS agentName, pr.name as productName
FROM Orders o INNER JOIN People pa ON o.agentId = pa.pid
              INNER JOIN People pc ON o.custId = pc.pid
              INNER JOIN Products pr ON o.prodId = pr.prodId
WHERE pa.homeCity = 'Regina';
-- AI
SELECT pc.firstName || ' ' || pc.lastName AS customer_name,
pr.name AS product_name,
pa.firstName || ' ' || pa.lastName AS agent_name
FROM Orders o
JOIN Customers c ON o.custId = c.pid
JOIN People pc ON c.pid = pc.pid
JOIN Products pr ON o.prodId = pr.prodId
JOIN Agents a ON o.agentId = a.pid
JOIN People pa ON a.pid = pa.pid
WHERE pa.homeCity = 'Regina';
```

Grade: 10/10; I'm not sure we needed all of those joins though...

6. Write a query to check the accuracy of the totalUSD column in the Orders table. This means calculating Orders.totalUSD from data in other tables and comparing those values to the values in Orders.totalUSD. Display all rows in Orders where Orders.totalUSD is incorrect, if any. If there are any incorrect values, explain why they are wrong. Round to exactly two decimal places.

```
-- Me
-- Orders 1017 and 1024 appear to be wrong due to a mistype. Order 1026 is wrong
because they mistakenly used Customer 10's discountPct (10.01%) instead of
Customer 7's (2%).
SELECT o.*, ROUND(o.quantityOrdered * pr.priceUSD * (1 - (c.discountPct * .01)),
2) as correctTotal
FROM Orders o INNER JOIN Products pr ON o.prodId = pr.prodId
              INNER JOIN Customers c ON c.pid = o.custId
```

```
WHERE o.totalUSD != ROUND(o.quantityOrdered * pr.priceUSD * (1 - (c.discountPct *
.01)), 2);
-- AI
SELECT o.*
FROM Orders o
JOIN Products p ON o.prodId = p.prodId
WHERE ROUND(o.quantityOrdered * p.priceUSD, 2) <> ROUND(o.totalUSD, 2);
```

Grade: 6/10; AI did not take into account the customer discount percent. That aside, it did well.

7. Display the first and last name of all customers who are also agents

```
-- Me
SELECT p.firstName, p.lastName
FROM People p INNER JOIN Agents a ON p.pid = a.pid
              INNER JOIN Customers c ON p.pid = c.pid;
-- AI
SELECT p.firstName, p.lastName
FROM People p
WHERE p.pid IN (SELECT pid FROM Customers)
AND p.pid IN (SELECT pid FROM Agents);
```

Grade: 10/10; I'm stuck in join mode now. And I thought I loved subqueries.

8. Create a VIEW of all Customer and People data called PeopleCustomers. Then another VIEW of all Agent and People data called PeopleAgents. Then select * from each of them to test them

```
-- Me
-- I won't be losing any bets!!
CREATE VIEW PeopleCustomers AS
SELECT
    c.pid,
    c.paymentTerms,
    c.discountPct,
    p.prefix,
    p.firstName,
    p.lastName,
    p.suffix,
    p.homeCity,
    p.DOB
FROM Customers c INNER JOIN People p ON p.pid = c.pid;

SELECT *
FROM PeopleCustomers;

CREATE VIEW PeopleAgents AS
SELECT
    a.pid,
    a.paymentTerms,
```

```
    a.commissionPct,
    p.prefix,
    p.firstName,
    p.lastName,
    p.suffix,
    p.homeCity,
    p.DOB
FROM Agents a INNER JOIN People p ON p.pid = a.pid;

SELECT *
FROM PeopleAgents;
-- AI
CREATE VIEW PeopleCustomersAI AS
SELECT p.*, c.paymentTerms, c.discountPct
FROM People p
JOIN Customers c ON p.pid = c.pid;

CREATE VIEW PeopleAgentsAI AS
SELECT p.*, a.paymentTerms, a.commissionPct
FROM People p
JOIN Agents a ON p.pid = a.pid;

SELECT * FROM PeopleCustomersAI;
SELECT * FROM PeopleAgentsAI;
```

Grade: 9.9/10. Gemini might lose a bet soon.

9. Display the first and last name of all customers who are also agents, this time using the views you created

```
-- Me
SELECT pc.firstName, pc.lastName
FROM PeopleCustomers pc INNER JOIN PeopleAgents pa ON pa.pid = pc.pid;
-- AI
SELECT pc.firstName, pc.lastName
FROM PeopleCustomers pc
WHERE pc.pid IN (SELECT pid FROM Agents);
```

Grade: 10/10; I guess table joins really did grow on me.

10. Compare your SQL in #7 (no views) and #9 (using views). The output is the same. How does that work? What is the database server doing internally when it processes the #9 query?

When a query uses a view, the database server replaces the view name with the view's query definition. The query with views becomes a query that utilizes the view's query to access underlying tables. Making a view is a great way to abstract queries or reduce complexity. As discussed in class, views can help greatly to reduce complexity in cases such as using middleware that cannot process complex queries. One caveat to note though is that a view made in postgres will explicitly reference the columns present when the view was

created. Even if you specified select * in the query, adding viewing is source code will reveal the enumeration. If another column is added, it will not be present!

11. [Bonus] What's the difference between a LEFT OUTER JOIN and a RIGHT OUTER JOIN? Give example queries in SQL to demonstrate. (Feel free to use the CAP database to make your points here).

Left outer join: all rows from the 'left' table, and the matched rows from the 'right' table. If there's no match, a NULL value is returned.

```
SELECT p.firstName, c.paymentTerms
FROM People p
LEFT OUTER JOIN Customers c ON p.pid = c.pid;
```

This query will return **all** people, even if they are not customers. For those who are not customers, paymentTerms will be NULL (no match).

Right outer join: all rows from the 'right' table, and the matched rows from the 'left' table (opposite of left outer join!). If there's no match, NULL is returned.

```
SELECT p.firstName, c.paymentTerms
FROM Customers c
RIGHT OUTER JOIN People p ON c.pid = p.pid;
```

This query will return **all** people, even if they are not customers. For those who are not customers, paymentTerms will be NULL. In this case, since a customer pid must exist in the people table, the result is the same as an inner join with nulls. The right join is useful when you want to make sure all records from the right table are returned, even if there are no matching records in the left table.