Capstone Project Lab

July 22, 2025

1 Ungraded Lab: Capstone Project Lab

1.1 Overview

Welcome to the Capstone Project Lab! In this comprehensive hands-on session, you'll apply all the SQL concepts you've learned throughout the course to analyze a complex dataset from TechMart, a growing retail chain. You'll clean data, write advanced queries, and produce a well-documented analysis report. This lab simulates real-world data analysis challenges, preparing you for your future career as a data scientist.

1.2 Learning Outcomes

By the end of this lab, you will be able to: - Clean and prepare complex datasets using SQL - Write advanced SQL queries involving subqueries, CTEs, and window functions - Perform comprehensive data analysis across multiple related tables - Leverage generative AI tools to optimize SQL queries and enhance performance - Produce a well-documented data analysis report

1.3 Dataset Information

You'll be working with the TechMart dataset, which contains information about a retail chain's operations across North America and Europe. The dataset includes: - Employee_Records: Information about employees, their roles, and sales performance - Product_Details: Details about products, including categories and inventory - Customer_Demographics: Customer information and loyalty program status - Sales_Transactions: Transaction data linking customers, products, and employees

1.4 Activities

1.4.1 Activity 1: Data Exploration and Cleaning

Before diving into analysis, it's crucial to understand and clean our dataset. We'll start by examining each table and addressing any data quality issues.

Step 1: Connect to the database, then load and display tables:

[1]:
```
import sqlite3
import pandas as pd

# Setting up the database. DO NOT edit the code given below
from techsmart_db_setup import setup_database
```

```
setup_database()
conn = sqlite3.connect('techsmart.db')

# Load and display tables
tables = ['Employee_Records', 'Product_Details', 'Customer_Demographics',␣
 ↪'Sales_Transactions']
for table in tables:
    query = f"SELECT * FROM {table} LIMIT 5"
    df = pd.read_sql_query(query, conn)
    print(f"\n{table}:")
    display(df)
```

```
Database setup complete: Tables created and populated with data!

Employee_Records:

   employee_id        role store_location sales_performance
0            1     Cashier       New York              None
1            2     Cashier    Los Angeles              3000
2            3  Supervisor        Phoenix              None
3            4     Manager        Chicago              3000
4            5     Manager        Phoenix              3000


Product_Details:

   product_id product_name     category        price  stock
0         106       Tablet  Electronics  one hundred  200.0
1         101     Keyboard  Accessories        fifty  100.0
2         108     Keyboard  Accessories        fifty  500.0
3         104      Speaker  Electronics          500  150.0
4         107      Speaker  Electronics        fifty  300.0


Customer_Demographics:

   customer_id   age gender     location loyalty_program
0            1    35      F  San Antonio            None
1            2    50      M       London              No
2            3    37      M       Austin              No
3            4  None      M  San Antonio            None
4            5    28      M  San Antonio            None


Sales_Transactions:

   transaction_id  customer_id product_id  employee_id quantity  total_amount  \
0               1         13.0        105          2.0        4          90.0
1               2         37.0        105         51.0        2          90.0
2               3          5.0       106a          NaN    three           NaN
3               4         35.0       109b         22.0    three          40.0
```

```
4                5         15.0        101        45.0        2           NaN

            sale_date
0  2025-03-03 00:00:00
1  2025-03-08 00:00:00
2  2025-03-01 00:00:00
3  2025-03-04 00:00:00
4  2025-03-06 00:00:00
```

Step 2: Identify and handle missing values:

[2]:
```
# Example for Employee_Records
query = """
SELECT COUNT(*) as total_rows,
       SUM(CASE WHEN sales_performance IS NULL OR sales_performance = 'nan'␣
 ↪THEN 1 ELSE 0 END) as missing_sales
FROM Employee_Records
"""
df = pd.read_sql_query(query, conn)
display(df)
```

```
   total_rows  missing_sales
0         100             11
```

Step 3: Try it yourself: Write queries to identify missing values in other tables

[3]:
```
# Your turn: Write queries to identify missing values in other tables
query = """
SELECT 'Product_Details' AS table_name,
       COUNT(*) AS total_rows,
       SUM(CASE WHEN price IS NULL OR TRIM(price) IN ('', 'nan', 'N/A') OR NOT␣
 ↪price GLOB '[0-9]*[.]?[0-9]*' THEN 1 ELSE 0 END) AS invalid_prices,
       SUM(CASE WHEN stock IS NULL OR TRIM(stock) IN ('', 'nan', 'N/A') OR NOT␣
 ↪stock GLOB '[0-9]*' THEN 1 ELSE 0 END) AS invalid_stock_levels
FROM Product_Details

UNION ALL

SELECT 'Customer_Demographics' AS table_name,
       COUNT(*) AS total_rows,
       SUM(CASE WHEN age IS NULL OR TRIM(age) IN ('', 'nan', 'N/A') OR NOT age␣
 ↪GLOB '[0-9]*' THEN 1 ELSE 0 END) AS invalid_ages,
       SUM(CASE WHEN loyalty_program IS NULL OR TRIM(loyalty_program) IN ('',␣
 ↪'nan', 'N/A') THEN 1 ELSE 0 END) AS invalid_loyalty_entries
FROM Customer_Demographics

UNION ALL
```

```
SELECT 'Sales_Transactions' AS table_name,
       COUNT(*) AS total_rows,
       SUM(CASE WHEN quantity IS NULL OR TRIM(quantity) IN ('', 'nan', 'N/A')␣
↪OR NOT quantity GLOB '[0-9]*' THEN 1 ELSE 0 END) AS invalid_quantity,
       SUM(CASE WHEN total_amount IS NULL OR TRIM(total_amount) IN ('', 'nan',␣
↪'N/A') OR NOT total_amount GLOB '[0-9]*[.]?[0-9]*' THEN 1 ELSE 0 END) AS␣
↪invalid_total_amount
FROM Sales_Transactions;

"""
df = pd.read_sql_query(query, conn)
display(df)
```

	table_name	total_rows	invalid_prices	invalid_stock_levels
0	Product_Details	100	100	19
1	Customer_Demographics	100	16	35
2	Sales_Transactions	100	32	100

Step 4: Clean inconsistent data formats:

[4]:
```
# Example: Standardize sales_performance in Employee_Records
query = """
UPDATE Employee_Records
SET sales_performance = CASE
    WHEN sales_performance = 'nan' THEN NULL
    WHEN sales_performance = 'five thousand' THEN '5000'
    ELSE sales_performance
END
"""
cursor = conn.cursor()
cursor.execute(query)
conn.commit()
```

Step 5: Try it yourself: Clean inconsistent data in other tables

[5]:
```
query = """
-- All updates combined
UPDATE Product_Details
SET price = CASE
    WHEN price = 'one hundred' THEN '100'
    WHEN price = 'fifty' THEN '50'
    ELSE price
END;

UPDATE Customer_Demographics
SET age = NULL
WHERE age IN ('None', 'nan', 'N/A');
```

```
UPDATE Customer_Demographics
SET loyalty_program = NULL
WHERE loyalty_program IN ('None', 'nan', 'N/A');

UPDATE Sales_Transactions
SET quantity = CASE
    WHEN quantity = 'three' THEN '3'
    ELSE quantity
END;

UPDATE Sales_Transactions
SET product_id = REPLACE(product_id, 'a', '');

UPDATE Sales_Transactions
SET product_id = REPLACE(product_id, 'b', '');

UPDATE Sales_Transactions
SET total_amount = NULL
WHERE total_amount IN ('nan', 'None', '');
"""

# Run cleaning updates
cursor = conn.cursor()
cursor.executescript(query)
conn.commit()

#  Follow up with a SELECT to verify cleanup (optional)
verify_query = """
SELECT * FROM Product_Details LIMIT 5;
"""
df = pd.read_sql_query(verify_query, conn)
display(df)
```

```
   product_id product_name     category price  stock
0         106       Tablet  Electronics   100  200.0
1         101     Keyboard  Accessories    50  100.0
2         108     Keyboard  Accessories    50  500.0
3         104      Speaker  Electronics   500  150.0
4         107      Speaker  Electronics    50  300.0
```

Tip: Use CASE statements to handle multiple conditions when cleaning data.

1.4.2 Activity 2: Advanced Data Analysis

Now that our data is clean, let's perform some advanced analysis to gain insights into TechMart's operations.

Step 1: Analyze employee performance by location:

[6]:

```
query = """
WITH emp_sales AS (
    SELECT store_location,
           AVG(CAST(sales_performance AS FLOAT)) as avg_sales,
           COUNT(*) as employee_count
    FROM Employee_Records
    WHERE sales_performance IS NOT NULL
    GROUP BY store_location
)
SELECT store_location, avg_sales, employee_count,
       RANK() OVER (ORDER BY avg_sales DESC) as location_rank
FROM emp_sales
ORDER BY avg_sales DESC
"""
df = pd.read_sql_query(query, conn)
display(df)
```

	store_location	avg_sales	employee_count	location_rank
0	Chicago	3875.000000	8	1
1	Phoenix	3312.500000	16	2
2	London	3214.285714	14	3
3	New York	2823.529412	17	4
4	Los Angeles	2692.307692	13	5
5	Paris	2600.000000	10	6
6	Houston	2363.636364	11	7

Step 2: Identify top-selling products by category:

[7]:

```
query = """
SELECT *
FROM (
    SELECT p.category,
           p.product_name,
           SUM(s.quantity) AS total_sold,
           RANK() OVER (
               PARTITION BY p.category
               ORDER BY SUM(s.quantity) DESC
           ) AS rank_in_category
    FROM Sales_Transactions s
    JOIN Product_Details p ON s.product_id = p.product_id
    GROUP BY p.category, p.product_name
)
WHERE rank_in_category <= 3
ORDER BY category, total_sold DESC
"""
df = pd.read_sql_query(query, conn)
display(df)
```

	category	product_name	total_sold	rank_in_category
0	Accessories	Charger	219	1
1	Accessories	Keyboard	211	2
2	Accessories	Monitor	110	3
3	Electronics	Tablet	268	1
4	Electronics	Mouse	180	2
5	Electronics	Headphones	125	3

Step 3: Try it yourself Analyze customer purchasing behavior:

[8]:

```
# Your turn: Write a query to analyze customer purchasing behavior
# Hint: Join Customer_Demographics with Sales_Transactions and use window␣
↪functions

query = """
WITH CustomerSales AS (
    SELECT
        c.customer_id,
        c.gender,
        c.location,
        c.age,
        c.loyalty_program,
        COUNT(t.transaction_id) AS total_transactions,
        SUM(CAST(t.total_amount AS FLOAT)) AS total_spent
    FROM Customer_Demographics c
    JOIN Sales_Transactions t ON c.customer_id = t.customer_id
    WHERE t.total_amount IS NOT NULL
    GROUP BY c.customer_id
),
RankedByCity AS (
    SELECT *,
           RANK() OVER (PARTITION BY location ORDER BY total_spent DESC) AS␣
↪spend_rank_in_city
    FROM CustomerSales
)
SELECT * FROM RankedByCity
ORDER BY total_spent DESC;

"""
df = pd.read_sql_query(query, conn)
display(df)
```

	customer_id	gender	location	age	loyalty_program	total_transactions	\
0	13	F	London	37	None	8	
1	12	M	San Antonio	37	Yes	6	
2	55	F	San Jose	40	None	6	
3	51	F	San Antonio	43	None	5	
4	46	F	Phoenix	20	No	6	

```
5               4      M  San Antonio  None          None              6
6              38      F  San Antonio    55            No              6
7              37      F      Chicago    55          None              6
8               5      M  San Antonio    28          None              5
9               3      M       Austin    37            No              4
10              2      M       London    50            No              4
11             15      F      Houston    30           Yes              4
12             41      M      Houston    22          None              4
13              1      F  San Antonio    35          None              3
14             42      M     New York    25           Yes              2
15             35      M  Los Angeles    42           Yes              2
16             10      F       Austin    41          None              1

    total_spent  spend_rank_in_city
0         440.0                   1
1         400.0                   1
2         350.0                   1
3         300.0                   2
4         290.0                   1
5         290.0                   3
6         290.0                   3
7         280.0                   1
8         270.0                   5
9         220.0                   1
10        210.0                   2
11        190.0                   1
12        180.0                   2
13        180.0                   6
14        150.0                   1
15         80.0                   1
16         50.0                   2
```

1.4.3 Activity 3: Performance Optimization

As our dataset grows, query performance becomes crucial. Let's optimize some of our complex queries. The below query analyzes sales performance for Electronics and Accessories by summarizing transactions per employee, store, and customer loyalty status, while also computing total revenue per store-category pair for ranking and comparison.

Step 1: Identify slow-running queries:

[9]:
```python
# Example: Time a complex query
import time

start_time = time.time()
query = """
WITH StoreSales AS (
   SELECT
```

```
        e.store_location,
        e.employee_id,
        e.role,
        p.category,
        c.loyalty_program,
        COUNT(s.transaction_id) AS total_sales,
        SUM(s.quantity) AS total_units_sold,
        SUM(s.total_amount) AS total_revenue
    FROM Sales_Transactions s
    JOIN Employee_Records e ON s.employee_id = e.employee_id
    JOIN Product_Details p ON s.product_id = p.product_id
    JOIN Customer_Demographics c ON s.customer_id = c.customer_id
    WHERE p.category IN ('Electronics', 'Accessories')
    GROUP BY e.store_location, e.employee_id, e.role, p.category, c.
 ↪loyalty_program
),

StoreRankings AS (
    SELECT
        store_location,
        category,
        SUM(total_revenue) AS store_revenue
    FROM StoreSales
    GROUP BY store_location, category
)

SELECT
    ss.store_location,
    ss.employee_id,
    ss.role,
    ss.category,
    ss.loyalty_program,
    ss.total_sales,
    ss.total_units_sold,
    ss.total_revenue
FROM StoreSales ss
JOIN StoreRankings sr ON ss.store_location = sr.store_location AND ss.category␣
 ↪= sr.category
ORDER BY ss.store_location, ss.total_revenue DESC;

"""


df = pd.read_sql_query(query, conn)
end_time = time.time()
print(f"Query execution time: {end_time - start_time} seconds")
```

```
Query execution time: 0.023155689239501953 seconds
```

Step 2: Add an index to optimize your query:

[10]:
```python
# Create an index
cursor.execute("CREATE INDEX idx_product_category ON Product_Details(category)")
conn.commit()
```

Step 3: Use an AI to further optimize your query:

Using an AI of your choice, further optimize your query then paste your updated query into the cell in Step 4.

Step 4: Try it yourself: Re-run your query and compare execution time

[11]:
```python
# Your Turn: Run your optimized query and compare execution time

start_time = time.time()
query = """
WITH FilteredProducts AS (
    SELECT product_id, category
    FROM Product_Details
    WHERE category IN ('Electronics', 'Accessories')
),

SalesData AS (
    SELECT
        s.transaction_id,
        s.quantity,
        s.total_amount,
        s.employee_id,
        s.product_id,
        s.customer_id
    FROM Sales_Transactions s
    JOIN FilteredProducts fp ON s.product_id = fp.product_id
    WHERE s.total_amount IS NOT NULL
),

StoreSales AS (
    SELECT
        e.store_location,
        e.employee_id,
        e.role,
        fp.category,
        c.loyalty_program,
        COUNT(sd.transaction_id) AS total_sales,
        SUM(CAST(sd.quantity AS INTEGER)) AS total_units_sold,
        SUM(CAST(sd.total_amount AS FLOAT)) AS total_revenue
    FROM SalesData sd
```

```
    JOIN Employee_Records e ON sd.employee_id = e.employee_id
    JOIN Customer_Demographics c ON sd.customer_id = c.customer_id
    JOIN FilteredProducts fp ON sd.product_id = fp.product_id
    GROUP BY e.store_location, e.employee_id, e.role, fp.category, c.
 ↪loyalty_program
),

StoreRankings AS (
    SELECT
        store_location,
        category,
        SUM(total_revenue) AS store_revenue
    FROM StoreSales
    GROUP BY store_location, category
)

SELECT
   ss.store_location,
   ss.employee_id,
   ss.role,
   ss.category,
   ss.loyalty_program,
   ss.total_sales,
   ss.total_units_sold,
   ss.total_revenue
FROM StoreSales ss
JOIN StoreRankings sr
    ON ss.store_location = sr.store_location
    AND ss.category = sr.category
ORDER BY ss.store_location, ss.total_revenue DESC;

"""

df = pd.read_sql_query(query, conn)
end_time = time.time()
print(f"Query execution time: {end_time - start_time} seconds")
```

```
Query execution time: 0.03339505195617676 seconds
```

Tip: Indexes can significantly improve query performance, but they also have overhead. Use them judiciously.

1.4.4 Activity 4: Generating the Analysis Report

Now that we've performed our analysis, it's time to compile our findings into a comprehensive report.

Step 1: Summarize key findings: - List the top 3 insights from your analysis - Provide supporting data for each insight

Step 2: Include SQL queries: - For each key insight, provide the SQL query used

Step 3: Document your process: - Explain your data cleaning steps - Describe any challenges you encountered and how you overcame them - Discuss potential areas for further analysis

Close the Connection It's good practice to close the database connection when you're done

```
[12]: # Close the database connection
conn.close()
```

1.5 Success Checklist

- Cleaned and prepared all dataset tables
- Performed at least 3 advanced SQL queries using subqueries, CTEs, or window functions
- Optimized at least one complex query for better performance
- Compiled a comprehensive analysis report with key insights and supporting data
- Program runs without errors

1.6 Common Issues & Solutions

- Problem: Query returns no results
  - Solution: Double-check table and column names, and ensure your JOIN conditions are correct
- Problem: Error "no such table"
  - Solution: Verify that you're connected to the correct database and that the table name is spelled correctly

1.7 Summary

In this comprehensive lab, you've applied advanced SQL concepts to analyze TechMart's retail operations data, working with multiple related tables covering employee records, product details, customer demographics, and sales transactions. You've gained hands-on experience in data cleaning, writing complex queries using subqueries, CTEs, and window functions, and optimizing query performance through indexing. Through this real-world simulation, you've developed the practical skills needed to conduct thorough data analysis and create well-documented reports, preparing you for actual data science roles.

1.7.1 Key Points

- Data cleaning is crucial for accurate analysis
- Advanced SQL techniques like CTEs and window functions enable complex analysis
- Query optimization is essential for working with large datasets
- Effective reporting is key to communicating insights