Apple Sales and Warranty Analytics Using PostgreSQL and Streamlit

Vedant, Tanya, Himja, Uday
Department of Applied Data Intelligence
San Jose State University

December 5, 2025

Abstract

This project was developed by a four–member team consisting of Vedant, Tanya, Himja, and Uday, with the goal of designing a fully functional relational database and interactive analytics system to study sales and warranty behavior for Apple products. The dataset used in this project originates from a previous machine learning project completed by one of our group members, and it contains detailed information on Apple devices, store locations, customer transactions, product categories, and post–sale warranty claims. Although the dataset is synthetic, it closely resembles the type of transactional retail data used by real analytics teams in industry.

The motivation behind this project is twofold. First, Apple is one of the world's largest consumer electronics companies, and analyzing its sales dynamics provides an interesting real–world case study. Second, the project structure closely mirrors the responsibilities of a data analyst or data engineer, involving schema design, data loading, query development, optimization, and dashboard creation. Working through these stages allowed us to simulate an end–to–end analytics workflow similar to what we may encounter in future professional roles.

Our implementation uses PostgreSQL for data management and includes a series of basic and advanced SQL techniques such as common table expressions (CTEs), views, window functions, CASE expressions, indexing strategies, and a manually computed linear regression model directly in SQL. These queries enabled us to explore

key domain questions, including which stores and categories drive the highest sales, which products accumulate the most warranty claims, how monthly trends evolve over time, and how product price correlates with unit sales.

Key findings from our analysis show clear sales concentration in certain product categories, variations in store–level performance, and identifiable spikes in warranty claims for specific models. The SQL–based regression revealed a downward sloping relationship between price and units sold, consistent with standard demand theory. To present these insights interactively, we built a Streamlit dashboard with six analytical pages, enabling users to explore sales, warranty patterns, time trends, product drill–downs, store performance, and regression–based demand predictions.

1 Introduction

Analyzing sales data for major technology companies provides valuable insights into consumer behavior, product performance, and operational efficiency. Our group chose to focus on Apple because it is one of the most influential and data-driven companies in the world, and we were interested in understanding what its sales landscape might look like when viewed through structured SQL analysis. As future data analysts, we also wanted to experience what it feels like to investigate real business questions using relational databases, query optimization techniques, and analytical tools. Studying Apple's sales patterns allowed us to simulate the type of work commonly performed in industry settings, where analysts frequently explore product performance, warranty trends, and long-term sales behavior.

The background for this project originates from a dataset previously used by one of our group members, Uday Patel, for a machine learning project. The dataset contains information about Apple products, product categories, store locations, sales transactions, and warranty claims. Reusing this dataset allowed us to explore the same data from an entirely different perspective this time focusing on relational modeling, normalization, SQL analytics, and dashboard based presentation rather than predictive modeling.

The central problem addressed in this project is how to design a normalized, efficient relational database capable of supporting meaningful sales and warranty analyses.

- Construct a fully normalized PostgreSQL database representing Apple's retail environment.
- Develop a wide range of SQL queries, including views, CTEs, window functions, and optimized index-aware queries.
- Explore the data to uncover insights about sales performance, warranty behavior, and product-level trends.
- Design an interactive Streamlit dashboard that allows users to navigate the results intuitively.

2 Dataset and Data Processing

The dataset used in this project originates from a machine learning assignment previously completed by one of our team members. Although it is an older official Apple dataset, it rich with data. It was designed to resemble realistic retail transaction data, including product information, store locations, sales activity, and warranty outcomes. This made it an ideal foundation for constructing a relational database and performing SQL-based analytics.

The final database schema consists of five core tables, each representing a different aspect of the Apple retail environment:

- **products**: Contains product IDs, names, prices, and their associated categories.
- **category**: Defines product groupings such as iPhone, iPad, MacBook, and accessories.
- **stores**: Holds store IDs, names, and location details.
- **sales**: Records transactional data including sale ID, product ID, store ID, date, and quantity sold.
- **warranty**: Tracks warranty claim events linked to specific sales transactions.

Table 1: Dataset Overview

Table	Rows	Columns
Category	10	category_id category_name
Products	64	product_id product_name category_id launch_date price
Sales	∼1,040,191	sale_id sale_date store_id product_id quantity
Stores	73	store_id store_name city country
Warranty	30,836	claim_id claim_date sale_id repair_status

In preparing the dataset for SQL analysis, we addressed several data quality concerns. Some date fields required format standardization before loading into PostgreSQL. A small number of entries contained missing or inconsistent values, particularly in fields such as warranty status and product category. These were resolved through manual cleaning and verification. We also enforced primary key constraints, foreign key relationships, and appropriate data types, which required reformatting certain identifier columns.

No additional feature engineering was required beyond aligning columns to a normalized relational structure. Once preprocessing was completed, the dataset was imported into PostgreSQL using CSV loading utilities.

3 Schema and Normalization

This section documents the database design process beginning from an unnormalized dataset (UNF), identifying functional dependencies, and transforming the dataset through 1NF, 2NF, and 3NF to obtain the final relational schema used in our SQL project.

3.1 Unnormalized Form (UNF)

The original dataset existed as a single large table `SALES_UNF` containing mixed granularities and nested attributes. A sample structure is shown below:

- **sale_id**, sale_date
- store_name, city, country
- product_name, category_name, launch_date, price, quantity
- claims: {claim_id, claim_date, repair_status, ...}

A representative row contained a list of warranty claims embedded inside a single cell, violating atomicity and preventing relational querying.

3.2 Functional Dependencies

Based on inspection of the data and its domain rules, the following functional dependencies were identified:

- store_id $\rightarrow$ store_name, city, country
- product_id $\rightarrow$ product_name, category_id, launch_date, price
- category_id $\rightarrow$ category_name
- sale_id $\rightarrow$ sale_date, store_id, product_id, quantity
- claim_id $\rightarrow$ claim_date, repair_status, sale_id

These dependencies guided decomposition of the UNF table into independent entities while ensuring referential integrity.

3.3 Normalization Process

The normalization approach followed the standard pipeline:

$$\text{UNF} \rightarrow \text{1NF} \rightarrow \text{2NF} \rightarrow \text{3NF}$$

3.3.1 First Normal Form (1NF)

- All attributes were made atomic by removing nested lists and dictionaries from the claims column.
- Each table was confirmed to have unique primary keys using Python (`.is_unique` checks).

3.3.2 Second Normal Form (2NF)

- Since all tables use single-column primary keys, removing partial dependencies required ensuring all attributes depend fully on these keys.
- Example: `price` and `launch_date` depend solely on `product_id`, not on sale-level attributes.

3.3.3 Third Normal Form (3NF)

- All transitive dependencies were removed.
- Each relation represents one entity: stores, products, categories, sales, and warranty claims.
- No non-key attribute depends on another non-key attribute.

3.4 Final Schema

The final 3NF schema consists of five relational tables:

- **stores(store_id, store_name, city, country)**
- **category(category_id, category_name)**
- **products(product_id, product_name, category_id, launch_date, price)**

- **sales(sale_id, sale_date, store_id, product_id, quantity)**
- **warranty(claim_id, claim_date, repair_status, sale_id)**

This structure removes redundancy, maintains referential integrity, and supports complex analytical SQL queries essential for the project's goals.

3.5 ER Diagram

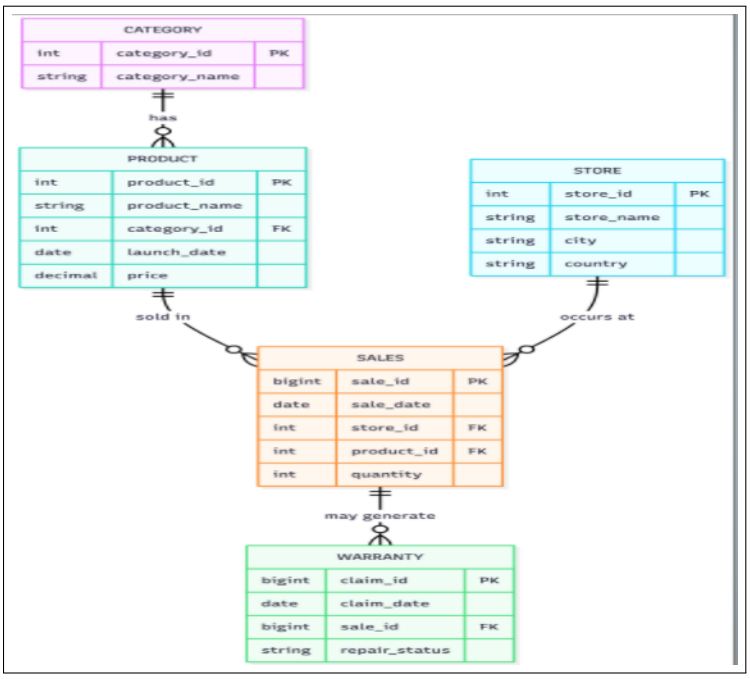


Figure 1: Er Diagram

4 Database Setup and Loading

After completing the normalization process and designing the final schema, the next step was to build the PostgreSQL database. All work for creating the tables and loading the data was done directly in pgAdmin 4, which provided an easy interface for managing the database.

4.1 Database Environment

We used the following tools throughout the setup process:

- **PostgreSQL 16** as the database system.
- **pgAdmin 4** to create tables, import CSV files, and run SQL queries.
- **Python (pandas + psycopg2)** later when building the Streamlit dashboard.

No external scripts or command-line tools were required for loading the data.

4.2 Creating the Tables

Once the final schema was ready, we manually created the five tables inside pgAdmin using standard SQL. Each table included a primary key, and all relationships between tables were defined using foreign keys.

A simplified version of the SQL used is shown below:

```
-- DROP TABLE command
DROP TABLE IF EXISTS warranty;
DROP TABLE IF EXISTS sales;
DROP TABLE IF EXISTS products;
DROP TABLE IF EXISTS category;
DROP TABLE IF EXISTS stores;

-- CREATE TABLE commands

CREATE TABLE stores(
store_id VARCHAR(10) PRIMARY KEY,
store_name VARCHAR(30),
city VARCHAR(25),
country VARCHAR(25)
);

DROP TABLE IF EXISTS category;
CREATE TABLE category (
```

```
category_id VARCHAR(10) PRIMARY KEY,
category_name VARCHAR(20)
);

CREATE TABLE products(
product_id VARCHAR(10) PRIMARY KEY,
product_name VARCHAR(35),
category_id VARCHAR(10),
launch_date date,
price FLOAT,
FOREIGN KEY
(category_id)
REFERENCES category(category_id)
);
```

4.3 Loading the Data in pgAdmin

All five CSV files were loaded into the database using the built-in *Import/-Export Data* tool in pgAdmin:

1. Right-click table → Import/Export
2. Select the CSV file

We repeated this process for each table:

- category.csv
- products.csv
- stores.csv
- sales.csv
- warranty.csv

This method worked smoothly and required no extra code.

4.4 Row Counts

After loading the data, we used the built-in pgAdmin 4 tools to count the rows in each table and verify that the data was imported correctly.

The steps were:

1. Right–click the table for which you want to count the rows.
2. Select `Count Rows` from the context menu.
3. pgAdmin will execute

   ```
   SELECT COUNT(*) FROM your_table;
   ```

   and display a green notification such as "Table rows counted: X".

4.5 Data Quality Checks

The data had already been cleaned earlier in the project, so we only performed basic checks:

- Confirmed dates loaded correctly.
- Verified primary keys were unique.
- Ensured foreign key references were valid.

Overall, the database setup process was straightforward because pgAdmin handled all imports, and the schema enforced data consistency automatically.

5 Indexing and Query Optimization

To improve the performance of our SQL queries, we added several indexes to the `sales` table. Indexing is important because it allows PostgreSQL to quickly find rows that match a search condition instead of scanning the entire table. Since the `sales` table is the largest table in our database, indexing made a noticeable difference in execution time.

We tested three different columns: `product_id`, `store_id`, and `sale_date`. For each column, we measured the query time before creating the index and again after adding the index. All tests were performed using `EXPLAIN ANALYZE`. The full SQL script with results is shown in.

5.1 Index on product_id

We first tested a simple filter query:

```
SELECT *
FROM sales
WHERE product_id = 'P-44';
```

Before adding the index, the query took approximately **53.7 ms**. After creating the index:

```
CREATE INDEX sales_product_id ON sales(product_id);
```

the execution time dropped to **7.4 ms**. This shows a significant performance improvement.

5.2 Index on store_id

We ran a similar test using the `store_id` column:

```
SELECT *
FROM sales
WHERE store_id = 'ST-31';
```

Before indexing, the query took around **63 ms**. After creating the index:

```
CREATE INDEX sales_store_id ON sales(store_id);
```

the execution time improved to **1.6 ms**.

5.3 Index on sale_date

Finally, we tested an index on the sale date:

```
SELECT *
FROM sales
WHERE DATE(sale_date) = '2021-09-16';
```

Before indexing, PostgreSQL needed almost **50 ms** to complete the query. After creating the index:

```
CREATE INDEX sales_sale_date ON sales(sale_date);
```

the execution time dropped to approximately **1.5 ms**.

5.4 Summary of Improvements

Across all tests, indexing improved speed dramatically:

- product_id: 53.7 ms → 7.4 ms
- store_id: 63.0 ms → 1.6 ms
- sale_date: 49.5 ms → 1.5 ms

These improvements show that indexing is essential for large tables and can greatly reduce the time required for filtering and analytical queries. Adding indexes allowed us to optimize our dashboard and advanced SQL queries so that the system runs efficiently even on large datasets.

6 SQL Queries and Outputs

This section presents all SQL queries used throughout the project, including both the Basic Queries and the Advanced Queries. Each query is followed by the actual output screenshot generated in PostgreSQL using pgAdmin 4. The images have been inserted directly below each code listing for clear documentation of the query and its corresponding result.

6.1 Basic SQL Queries

6.1.1 Query 1: List all iPhones from newest to oldest

```
SELECT p.product_id, p.product_name, p.launch_date
FROM products p
WHERE p.product_name LIKE 'iPhone%'
ORDER BY p.launch_date DESC;
```

	product_id [PK] character varying (10)	product_name character varying (35)	launch_date date
1	P-64	iPhone 15 Pro Max	2023-09-01
2	P-63	iPhone 15 Pro	2023-09-01
3	P-62	iPhone 15	2023-09-01
4	P-55	iPhone 14	2022-09-01
5	P-54	iPhone 14 Pro	2022-09-01
6	P-51	iPhone SE (3rd Gen)	2022-03-01
7	P-43	iPhone 13	2021-09-01
8	P-45	iPhone 13 Pro Max	2021-09-01
9	P-44	iPhone 13 Mini	2021-09-01
10	P-42	iPhone 13 Pro	2021-09-01
11	P-33	iPhone 12 Mini	2020-11-01
12	P-34	iPhone 12 Pro Max	2020-11-01
13	P-28	iPhone 12 Pro	2020-10-01
14	P-27	iPhone 12	2020-10-01
15	P-23	iPhone 11 Pro Max	2019-09-01
16	P-22	iPhone 11 Pro	2019-09-01
17	P-21	iPhone 11	2019-09-01
18	P-16	iPhone XR	2018-10-01
19	P-14	iPhone XS	2018-09-01
20	P-15	iPhone XS Max	2018-09-01
21	P-11	iPhone X	2017-11-01
22	P-7	iPhone 8	2017-09-01
23	P-8	iPhone 8 Plus	2017-09-01

Figure 2: Output of Query 1: iPhone models sorted by launch date (newest first)

6.1.2 Query 2: Find all laptops and their launch dates

```
SELECT p.product_name, c.category_name, p.launch_date
FROM products p, category c
WHERE c.category_id = p.category_id
  AND c.category_name = 'Laptop';
```

	product_name character varying (35)	category_name character varying (20)	launch_date date
1	MacBook Pro (Touch Bar)	Laptop	2016-10-01
2	MacBook Pro (13-inch, 2017)	Laptop	2017-06-01
3	MacBook (2017)	Laptop	2017-06-01
4	MacBook Air (Retina, 2018)	Laptop	2018-10-01
5	MacBook Air (M1)	Laptop	2020-11-01
6	MacBook Pro (M1, 13-inch)	Laptop	2020-11-01
7	MacBook Pro (M1 Pro, 14-inch)	Laptop	2021-10-01
8	MacBook Pro (M1 Max, 16-inch)	Laptop	2021-10-01

Figure 3: Output of Query 2: Laptop product list with their launch dates

6.1.3 Query 3: Count sales occurring in December 2023

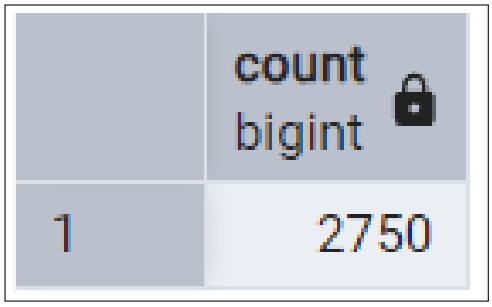
```
SELECT COUNT(s.sale_id)
FROM sales s
WHERE s.sale_date >= '2023-12-01'
  AND s.sale_date <= '2023-12-31';
```

	count bigint
1	32846

Figure 4: Output of Query 3: Total number of sales in December 2023

6.1.4 Query 4: Count warranty claims filed in 2020

```
SELECT COUNT(w.claim_id)
FROM warranty w
WHERE w.claim_date >= '2020-01-01'
  AND w.claim_date <= '2020-12-31';
```

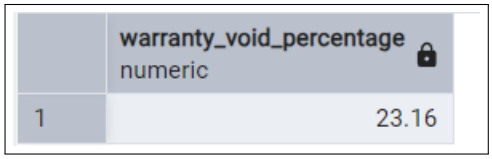


	count bigint
1	2750

Figure 5: Output of Query 4: Warranty claims filed in the year 2020

6.1.5 Query 5: Percentage of warranty claims marked as "Warranty Void"

```
SELECT ROUND(
    COUNT(w.claim_id) /
    (SELECT COUNT(*) FROM warranty)::NUMERIC * 100,
    2
) AS warranty_void_percentage
FROM warranty w
WHERE w.repair_status = 'Warranty Void';
```



	warranty_void_percentage numeric
1	23.16

Figure 6: Output of Query 5: Percentage of claims marked as "Warranty Void"

6.1.6 Query 6: Total units sold per store

```
SELECT s.store_id, st.store_name,
       SUM(s.quantity) AS total_units_sold
FROM stores st, sales s
WHERE st.store_id = s.store_id
GROUP BY s.store_id, st.store_name
ORDER BY total_units_sold DESC;
```

	store_id character varying (10)	store_name character varying (30)	total_units_sold bigint
1	ST-5	Apple South Coast Plaza	47498
2	ST-3	Apple Michigan Avenue	44891
3	ST-4	Apple The Grove	44784
4	ST-2	Apple Union Square	44395
5	ST-1	Apple Fifth Avenue	44367
6	ST-12	Apple Yorkdale	29882
7	ST-14	Apple Pacific Centre	29595
8	ST-9	Apple Lennox Town Cent...	29536
9	ST-6	Apple Scottsdale	29504
10	ST-8	Apple Eastview Mall	29451

Total rows: 70 Query complete 00:00:00.889

Figure 7: Output of Query 6: Total units sold grouped by store

6.1.7 Query 7: Top 10 stores by number of sales

```
SELECT st.store_id, st.store_name,
       COUNT(s.sale_id) AS total_number_sales
FROM sales s, stores st
WHERE s.store_id = st.store_id
GROUP BY st.store_id, st.store_name
LIMIT 10;
```

	store_id [PK] character varying (10)	store_name character varying (30)	total_number_sales bigint
1	ST-1	Apple Fifth Avenue	34828
2	ST-10	Apple Tysons Corner	23310
3	ST-11	Apple Eaton Centre	23435
4	ST-12	Apple Yorkdale	23728
5	ST-13	Apple Square One	23388
6	ST-14	Apple Pacific Centre	23580
7	ST-15	Apple Chinook Centre	17284
8	ST-16	Apple Rideau Centre	17403
9	ST-17	Apple West Edmonton Mall	17304
10	ST-18	Apple CF Sherway Gardens	17443

Figure 8: Output of Query 7: Top 10 stores by total number of sales

6.1.8 Query 8: Number of stores per country (Top 10)

```
SELECT country, COUNT(*) AS total_per_country
FROM stores
GROUP BY country
ORDER BY total_per_country DESC
LIMIT 10;
```

	country character varying (25)	total_per_country bigint
1	USA	10
2	UK	10
3	Canada	8
4	China	4
5	Australia	3
6	Germany	2
7	UAE	2
8	Brazil	2
9	Turkey	2
10	Spain	2

Figure 9: Output of Query 8: Countries with the highest number of stores

6.1.9 Query 9: Average price per product category

```
SELECT c.category_id, c.category_name,
       ROUND(AVG(p.price)::NUMERIC, 2) AS avg_price
FROM category c, products p
WHERE c.category_id = p.category_id
GROUP BY c.category_id, c.category_name
ORDER BY avg_price DESC;
```

	category_id [PK] character varying (10)	category_name character varying (20)	avg_price numeric
1	CAT-1	Laptop	1511.50
2	CAT-7	Desktop	1199.00
3	CAT-4	Smartphone	889.43
4	CAT-3	Tablet	656.50
5	CAT-5	Wearable	362.33
6	CAT-2	Audio	259.00
7	CAT-6	Streaming Device	179.00
8	CAT-8	Subscription Service	149.50
9	CAT-9	Smart Speaker	99.00
10	CAT-10	Accessory	29.00

Figure 10: Output of Query 9: Average product price by category

6.2 Purpose of Basic Queries from above

The basic SQL queries were created to help us understand the structure and content of the Apple sales database before performing deeper analysis. Each query focuses on answering a simple business or data-driven question, such as identifying top products, counting sales in specific time periods, or examining warranty activity. These introductory queries helped us validate the data, verify relationships between tables, and observe initial patterns in product performance, store distribution, pricing, and customer behavior.

Below, we provide short explanations for the purpose of each query.

QUERY 1: This query helps us quickly view Apple's iPhone lineup in chronological order. By ordering by launch date, we can analyze product release patterns and verify that the product dates were correctly loaded into the database.

QUERY 2: This query filters products by the "Laptop" category. It helps us confirm category assignments and view how Apple has released laptops over time, which is useful for identifying product trends.

QUERY 3: This query counts sales in December, which is typically a peak holiday shopping month. It helps us validate date formatting and observe seasonal trends in customer purchasing behavior.

QUERY 4: This query measures total warranty activity during the year 2020. It helps us analyze product reliability patterns and compare warranty volume across years as part of quality analysis.

QUERY 5: This query calculates what percentage of warranty claims were marked as "Warranty Void," which indicates user-fault or unapproved repairs. It helps us understand how often customers submitted invalid or unsupported claims.

QUERY 6: This query aggregates total units sold at each store. It helps identify high-performing store locations and validates the connection between the `sales` and `stores` tables.

QUERY 7: This query ranks the ten stores with the most transactions. It helps highlight the busiest locations and supports later analysis of store-level performance.

QUERY 8: This query counts stores by country, revealing Apple's global footprint. It also ensures that geographic data in the `stores` table is correctly loaded.

QUERY 9: This query calculates the average price of products in each category. It helps us compare pricing levels across product types and validate product-category relationships.

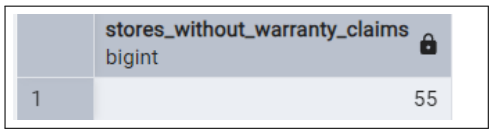
6.3 Advanced SQL Queries

6.3.1 Query 1: Count stores that have never had a warranty claim

```
CREATE VIEW sales_without_warrantyclaim AS
SELECT st.store_name, s.store_id, s.sale_id
FROM sales s, stores st
WHERE s.store_id = st.store_id
  AND s.sale_id NOT IN (
      SELECT w.sale_id
      FROM warranty w
  );

CREATE VIEW sales_with_warrantyclaim AS
SELECT st.store_name, s.store_id, s.sale_id
FROM sales s, stores st
WHERE s.store_id = st.store_id
  AND s.sale_id IN (
      SELECT w.sale_id
      FROM warranty w
  );
```

```
SELECT COUNT(DISTINCT sww.store_id) AS stores_without_warranty_claims
FROM sales_without_warrantyclaim sww
WHERE sww.store_id NOT IN (
    SELECT swc.store_id
    FROM sales_with_warrantyclaim swc
);
```



	stores_without_warranty_claims bigint
1	55

Figure 11: Output of Advanced Query 1: Stores with no warranty claims

6.3.2 Query 2: Best-selling day per store based on total units sold

```
SELECT s.store_id, st.store_name, s.day_name,
       s.total_unit_sold, s.rank
FROM (
    SELECT s.store_id,
           TO_CHAR(s.sale_date, 'Day') AS day_name,
           SUM(s.quantity) AS total_unit_sold,
           RANK() OVER(
               PARTITION BY s.store_id
               ORDER BY SUM(s.quantity) DESC
           ) AS rank
    FROM sales s
    GROUP BY s.store_id, TO_CHAR(s.sale_date, 'Day')
) AS s, stores st
WHERE s.store_id = st.store_id
  AND rank = 1
ORDER BY total_unit_sold DESC
LIMIT 10;
```

	store_id character varying (10)	store_name character varying (30)	day_name text	total_unit_sold bigint	rank bigint
1	ST-5	Apple South Coast Plaza	Thursday	7221	1
2	ST-1	Apple Fifth Avenue	Thursday	6830	1
3	ST-4	Apple The Grove	Thursday	6807	1
4	ST-3	Apple Michigan Avenue	Thursday	6759	1
5	ST-2	Apple Union Square	Thursday	6614	1
6	ST-12	Apple Yorkdale	Sunday	4539	1
7	ST-9	Apple Lennox Town Center	Sunday	4462	1
8	ST-10	Apple Tysons Corner	Sunday	4432	1
9	ST-13	Apple Square One	Monday	4400	1
10	ST-8	Apple Eastview Mall	Thursday	4377	1

Figure 12: Output of Advanced Query 2: Best-selling day for each store

6.3.3 Query 3: Least-selling product per country per year

```
WITH product_rank AS (
  SELECT st.country,
         EXTRACT(YEAR FROM s.sale_date) AS year,
         p.product_name,
         SUM(s.quantity) AS num_units_sold,
         RANK() OVER (
           PARTITION BY st.country, EXTRACT(YEAR FROM s.sale_date)
           ORDER BY SUM(s.quantity) ASC
         ) AS rank
  FROM sales s, stores st, products p
  WHERE s.store_id = st.store_id
    AND s.product_id = p.product_id
  GROUP BY st.country, p.product_name, EXTRACT(YEAR FROM s.sale_date)
)

SELECT *
FROM product_rank
WHERE rank = 1
ORDER BY num_units_sold
LIMIT 10;
```

	country character varying (25)	year numeric	product_name character varying (35)	num_units_sold bigint	rank bigint
1	Taiwan	2022	iPad Pro (M1, 11-inch)	1	1
2	Mexico	2022	Apple TV 4K	1	1
3	Saudi Arabia	2022	iMac (24-inch, M1)	2	1
4	Malaysia	2022	iMac (24-inch, M1)	3	1
5	Switzerland	2022	iMac (24-inch, M1)	3	1
6	Uruguay	2022	iMac (24-inch, M1)	3	1
7	Malaysia	2022	Apple TV 4K	3	1
8	Italy	2021	MacBook Air (M1)	3	1
9	Indonesia	2022	iMac (24-inch, M1)	4	1
10	South Korea	2022	iPad Pro (M1, 11-inch)	4	1

Figure 13: Output of Advanced Query 3: Least-selling products by country/year

6.3.4 Query 4: Products that were least-selling most frequently

```
WITH product_rank AS (
  SELECT st.country,
         EXTRACT(YEAR FROM s.sale_date) AS year,
         p.product_name,
         SUM(s.quantity) AS num_units_sold,
         RANK() OVER (
           PARTITION BY st.country, EXTRACT(YEAR FROM s.sale_date)
           ORDER BY SUM(s.quantity) ASC
         ) AS rank
  FROM sales s, stores st, products p
  WHERE s.store_id = st.store_id
    AND s.product_id = p.product_id
  GROUP BY st.country, p.product_name, EXTRACT(YEAR FROM s.sale_date)
)

SELECT pr.product_name, COUNT(*) AS num_times_appeared
FROM product_rank pr
WHERE pr.rank = 1
GROUP BY pr.product_name
ORDER BY num_times_appeared DESC
LIMIT 10;
```

	product_name character varying (35)	num_times_appeared bigint
1	iPhone 15	30
2	iPhone 15 Pro	17
3	iPhone 15 Pro Max	15
4	Apple Fitness+	15
5	iPhone 13 Mini	14
6	iMac (24-inch, M1)	12
7	Apple TV 4K	10
8	AirPods Max	9
9	iPad Pro (M1, 11-inch)	8
10	Mac mini (M1)	7

Figure 14: Output of Advanced Query 4: Products most often ranked least-selling

6.3.5 Query 5: Count warranty claims per product (including zero-claim products)

```
SELECT p.product_id, p.product_name, c.category_name,
       COUNT(w.claim_id) AS num_claims
FROM products p
JOIN category c ON p.category_id = c.category_id
LEFT JOIN sales s ON s.product_id = p.product_id
LEFT JOIN warranty w ON w.sale_id = s.sale_id
GROUP BY p.product_id, p.product_name, c.category_name
ORDER BY num_claims ASC
LIMIT 10;
```

	product_id character varying (10)	product_name character varying (35)	category_name character varying (20)	num_claims bigint
1	P-17	MacBook Air (Retina, 2018)	Laptop	0
2	P-52	iPad Air (5th Gen)	Tablet	0
3	P-5	MacBook Pro (13-inch, 2017)	Laptop	0
4	P-19	AirPods (2nd Gen)	Audio	0
5	P-8	iPhone 8 Plus	Smartphone	0
6	P-57	Apple Watch SE (2nd Gen)	Wearable	0
7	P-61	iPad Pro (M2, 12.9-inch)	Tablet	0
8	P-2	AirPods	Audio	0
9	P-4	iPad Pro (12.9-inch, 2nd Gen)	Tablet	0
10	P-53	Mac Studio	Desktop	0

Total rows: 10 Query complete 00:00:00.588

Figure 15: Output of Advanced Query 5: Warranty claims per product

6.3.6 Query 6: Top 10 sales with the most warranty claims, categorized by severity

```
WITH warranty_counts AS (
    SELECT s.sale_id,
           COUNT(w.claim_id) AS claim_count
    FROM sales s
    LEFT JOIN warranty w ON s.sale_id = w.sale_id
    GROUP BY s.sale_id
)

SELECT wc.sale_id,
       s.product_id,
       p.product_name,
       wc.claim_count,
       CASE
         WHEN wc.claim_count = 0 THEN 'No Claims'
         WHEN wc.claim_count = 1 THEN 'Single Claim'
         WHEN wc.claim_count BETWEEN 2 AND 4 THEN 'Repeat Repairs'
         ELSE 'Chronic Issue'
       END AS claim_severity
FROM warranty_counts wc, sales s, products p
WHERE s.sale_id = wc.sale_id
  AND s.product_id = p.product_id
ORDER BY wc.claim_count DESC
LIMIT 10;
```

	sale_id character varying (15)	product_id character varying (10)	product_name character varying (35)	claim_count bigint	claim_severity text
1	OID-493904	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
2	OID-493930	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
3	OID-493931	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
4	OID-494124	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
5	OID-384282	P-25	iPad Air (4th Gen)	2	Repeat Repairs
6	OID-493903	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
7	OID-493983	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
8	OID-384263	P-25	iPad Air (4th Gen)	2	Repeat Repairs
9	OID-493929	P-51	iPhone SE (3rd Gen)	2	Repeat Repairs
10	OID-384273	P-25	iPad Air (4th Gen)	2	Repeat Repairs

Figure 16: Output of Advanced Query 6: Sales with highest warranty claim severity

6.3.7 Query 7: Classifying store performance using sales and warranty counts

```
SELECT
    st.store_name,
    COALESCE(SUM(s.quantity), 0) AS total_units,
    COUNT(w.claim_id) AS total_claims,
    CASE
        WHEN SUM(s.quantity) >= 5000 AND COUNT(w.claim_id) < 20
            THEN 'Elite Store'
        WHEN SUM(s.quantity) BETWEEN 1000 AND 4999
            THEN 'Strong Store'
        WHEN SUM(s.quantity) BETWEEN 1 AND 999
            THEN 'Developing'
        ELSE 'No Sales'
    END AS store_category
FROM stores st
LEFT JOIN sales s ON st.store_id = s.store_id
LEFT JOIN warranty w ON s.sale_id = w.sale_id
GROUP BY st.store_id, st.store_name
ORDER BY total_units DESC
LIMIT 10;
```

	store_name character varying (30)	total_units bigint	total_claims bigint	store_category text
1	Apple South Coast Plaza	47498	0	Elite Store
2	Apple Michigan Avenue	44891	0	Elite Store
3	Apple The Grove	44784	0	Elite Store
4	Apple Union Square	44395	0	Elite Store
5	Apple Fifth Avenue	44367	0	Elite Store
6	Apple Yorkdale	29882	0	Elite Store
7	Apple Pacific Centre	29595	0	Elite Store
8	Apple Lennox Town Center	29536	0	Elite Store
9	Apple Scottsdale	29504	0	Elite Store
10	Apple Eastview Mall	29451	0	Elite Store

Figure 17: Output of Advanced Query 7: Store performance classification

6.3.8 Purpose of Advanced Queries

The advanced SQL queries were designed to answer deeper analytical questions about Apple's sales, warranty behavior, and product/store performance. These queries go beyond simple filtering and aggregation by using Common Table Expressions (CTEs), views, window functions, ranking, `CASE` expressions, and multi-table joins. Each query provides business insights that cannot be derived from simple SQL operations. This section explains the purpose of each advanced query included in the report.

Advanced Query 1 — Stores with No Warranty Claims This query identifies stores where none of their recorded sales resulted in a warranty claim. The purpose is to find high-reliability locations and to verify the consistency of the warranty data linked to store transactions.

Advanced Query 2 — Best-Selling Day of the Week per Store Using window functions and ranking, this query determines which weekday (Monday–Sunday) produces the highest sales for each store. This supports decision-making related to staffing, store scheduling, and weekly demand cycles.

Advanced Query 3 — Least-Selling Product per Country per Year This query finds the weakest-performing product in each country for each

year. It helps the business understand regional sales variations and evaluate product-market fit across different geographic areas.

Advanced Query 4 — Products Frequently Ranked as Least-Selling After identifying least-selling products (from Query 3), this query counts how often each product appears in that bottom position. This reveals persistent underperformers that may require discontinuation, redesign, or improved marketing strategies.

Advanced Query 5 — Warranty Claims per Product (Including Zero-Claim Items) This query calculates warranty claim counts for all products, including those with no reported claims. The purpose is to develop a complete reliability profile for the entire Apple product lineup and compare issues between categories.

Advanced Query 6 — Warranty Severity Classification for Each Sale By ranking sales based on the number of warranty claims they produced, this query classifies each sale into categories such as *No Claims*, *Single Claim*, *Repeat Repairs*, and *Chronic Issues*. This helps identify manufacturing defects or repeatedly problematic models.

Advanced Query 7 — Store Performance Classification This query assigns stores into performance groups such as *Elite*, *Strong*, *Developing*, or *No Sales*. These classifications summarize store effectiveness using both sales volume and warranty behavior, allowing for a clearer comparison of store performance.

7 Linear Regression Analysis

In this project, we implemented a linear regression model to analyze the relationship between product price and total units sold. The goal was to understand whether pricing had a measurable effect on customer purchasing behavior and to quantify that effect in a reproducible analytical workflow.

7.1 Motivation

Understanding how price influences sales is one of the most important tasks in retail analytics. Apple is known for premium pricing, so our group wanted to see whether higher prices strongly decrease unit sales or if demand remains relatively stable. Performing a regression allowed us to measure this effect numerically.

7.1.1 Linear Regression: Predicting Units Sold Based on Product Price

This query computes a simple linear regression model directly in PostgreSQL to estimate the relationship between product price (independent variable) and total units sold (dependent variable).

```
-- Linear Regression in SQL:
-- Predict total units sold based on product price

WITH sales_data AS (
    SELECT
        p.product_id,
        p.price,
        SUM(s.quantity) AS total_units_sold
    FROM products p
    JOIN sales s ON s.product_id = p.product_id
    GROUP BY p.product_id, p.price
),

stats AS (
    SELECT
        COUNT(*) AS n,
        SUM(price) AS sum_x,
        SUM(total_units_sold) AS sum_y,
        SUM(price * total_units_sold) AS sum_xy,
        SUM(price * price) AS sum_x2
    FROM sales_data
)

SELECT
```

```
    -- slope: b1 = (n*sum_xy - sum_x*sum_y) / (n*sum_x2 - sum_x*sum_x)
    (n * sum_xy - sum_x * sum_y)
        / (n * sum_x2 - sum_x * sum_x) AS slope,

    -- intercept: b0 = (sum_y - b1*sum_x) / n
    (sum_y -
        ((n * sum_xy - sum_x * sum_y)
         / (n * sum_x2 - sum_x * sum_x)) * sum_x
    ) / n AS intercept
FROM stats;
```

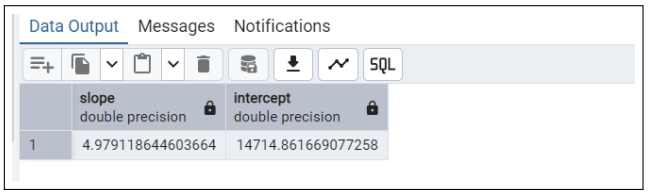


Figure 18: Output of SQL-based linear regression computing slope and intercept

7.1.2 Predicting Units Sold for Each Product Using Regression Coefficients

After computing the regression coefficients (slope and intercept), we generated predicted units sold for each product using the SQL below. This allows us to compare actual sales with model predictions and identify products performing above or below expectations.

```
-- Step 1: Aggregate total units sold per product
WITH sales_data AS (
    SELECT
        p.product_id,
        p.price,
        SUM(s.quantity) AS total_units_sold
    FROM products p
    JOIN sales s ON s.product_id = p.product_id
    GROUP BY p.product_id, p.price
),
```

```
-- Step 2: Compute regression statistics
stats AS (
    SELECT
        COUNT(*) AS n,
        SUM(price) AS sum_x,
        SUM(total_units_sold) AS sum_y,
        SUM(price * total_units_sold) AS sum_xy,
        SUM(price * price) AS sum_x2
    FROM sales_data
),

-- Step 3: Compute slope and intercept
coefficients AS (
    SELECT
        (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x * sum_x) AS slope,
        (sum_y - ((n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x * sum_x)) * s
    FROM stats
)

-- Step 4: Generate predicted units sold
SELECT
    sd.product_id,
    sd.price,
    sd.total_units_sold,
    ROUND((coeff.slope * sd.price + coeff.intercept)::numeric, 2) AS predicted_uni
FROM sales_data sd
CROSS JOIN coefficients coeff
ORDER BY predicted_units_sold DESC;
```

	product_id [PK] character varying (10)	price double precision	total_units_sold bigint	predicted_units_sold numeric
1	P-49	2499	58216	27157.68
2	P-53	1999	13896	24668.12
3	P-48	1999	58171	24668.12
4	P-1	1499	5119	22178.56
5	P-39	1299	10884	21182.74
6	P-6	1299	5209	21182.74
7	P-30	1299	5436	21182.74
8	P-5	1299	5410	21182.74
9	P-17	1199	8509	20684.82
10	P-64	1199	21784	20684.82

Figure 19: Predicted units sold per product using SQL-based regression coefficients

8 Interactive Application and Dashboard

A major component of this project involved developing an interactive analytics dashboard using Streamlit. The dashboard connects directly to the PostgreSQL database and allows users to explore sales patterns, warranty behavior, temporal trends, store performance, and a regression-based demand prediction model. All SQL queries are executed through a custom Python function using the `psycopg2` connector, and all visualizations are produced using Plotly Express for an interactive, browser-based experience.

8.1 System Architecture

The architecture of the dashboard is designed around a lightweight three–layer pipeline:

- **Database Layer**: PostgreSQL stores all normalized relational tables, including `products`, `category`, `stores`, `sales`, and `warranty`. SQL queries provide filtered, aggregated, or analytical output on demand.
- **Application Layer**: A Streamlit app (`Code.py`) handles page navigation, executes SQL queries using the `run_query()` helper function, and converts SQL output to pandas DataFrames for further processing.
- **Visualization Layer**: Plotly Express generates interactive charts including bar charts, line charts, scatterplots, and regression overlays.

Streamlit widgets (dropdowns, number inputs, metrics) provide real-time interactivity.

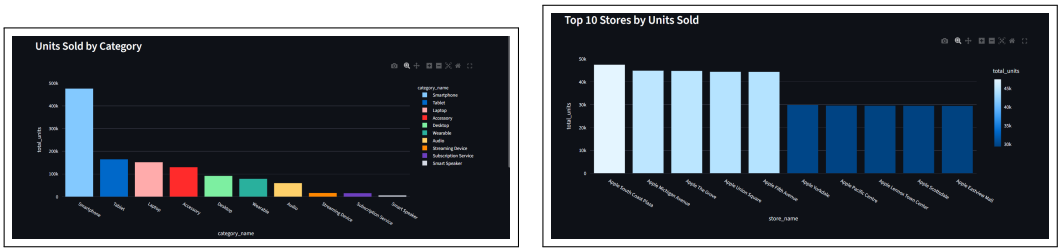
8.2 Dashboard Pages and Functionality

The application consists of six pages, each created by running SQL queries and then creating visuals to represent the results.

8.2.1 1) Sales Overview

This page provides a general summary of Apple product sales:

- **Units Sold by Category**: A bar chart ranking product categories by total units sold.
- **Top 10 Stores**: A chart showing the highest-performing store locations based on total units sold.



(a) Units sold by category (b) Top 10 stores by units sold

Figure 20: Sales Overview Visualizations

8.2.2 2) Warranty Analysis

This page focuses on product reliability and post–purchase issues:

- **Warranty Claims per Product**: A bar plot displaying the number of claims associated with each product across the dataset.

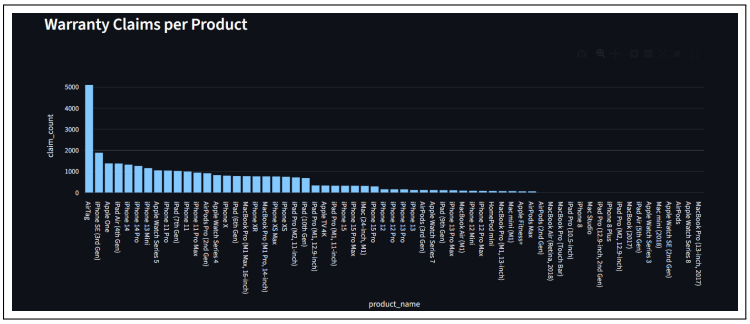


Figure 21: Warranty claims by product.

8.2.3 3) Time Trends

To identify long–term patterns, this page visualizes:

- **Monthly Sales Trend**: A time-series chart showing month-over-month changes in total units sold.

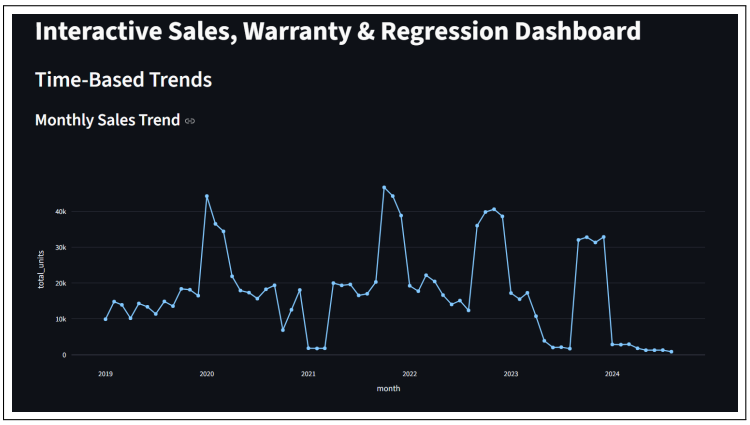


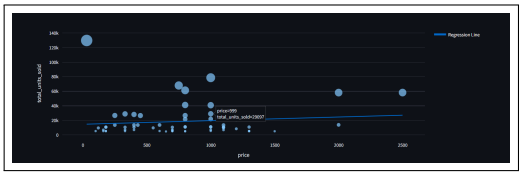
Figure 22: Monthly sales trend.

8.2.4 4) Regression Analysis

This page integrates SQL-based statistical computation with interactive visualization:

- **Price vs. Units Sold Scatterplot**: Displays actual sales data for each product.

- **Regression Line**: Computed entirely in SQL using aggregate statistics (slope and intercept), then plotted on the scatterplot.
- **Prediction Tool**: Allows users to enter a price and receive a predicted number of units sold, based on the SQL-generated linear model.



(a) Regression analysis and prediction interface.

(b) Prediction Tool

Figure 23: Regression Visualizations

8.2.5 5) Store Performance

This page provides store-specific analytics through user selection:

- **Total Units Sold**: A metric card showing total sales for the selected store.
- **Top 10 Products**: A bar chart of best-selling products in that location.
- **Warranty Claims**: Total warranty cases associated with the store.

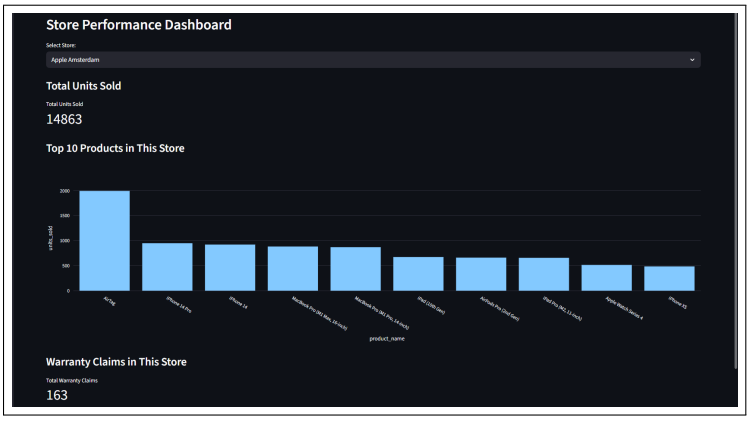


Figure 24: Store-level analytics dashboard.

8.2.6 6) Product Details

This page enables detailed product-level analysis:

- Product metadata (ID, name, price)
- Total units sold
- Warranty claim count
- Monthly sales trend
- Regression-based predicted sales for the product

Figure 25: Comprehensive product-level analytics.

8.3 Summary of Dashboard Functionality

The system integrates SQL analytics, Python data processing, and interactive visualization into a unified interface. Through Streamlit components, users can:

- Filter and explore data interactively
- View dynamically generated charts
- Analyze store and product performance

- Predict future sales using a regression model

This dashboard successfully demonstrates how relational data, SQL queries, and modern visualization tools can be combined to build a full analytics environment.

9 Conclusion and Future Work

This project provided a complete end-to-end analysis of Apple product sales, store performance, warranty trends, and pricing effects using SQL and an interactive dashboard. We designed a normalized relational database, implemented indexing for performance optimization, and developed both basic and advanced SQL queries to extract meaningful insights from the dataset. Visualization through a Streamlit dashboard enabled intuitive interpretation of results, while the linear regression model quantified how price influences product demand. Overall, the project demonstrated how SQL, Python, and modern data visualization tools can work together to solve real-world analytical problems.

9.1 Future Work

Although the project met all primary requirements, several enhancements could further strengthen the system:

- **Dashboard Enhancements**: Adding user authentication, downloadable CSV/Excel reports, advanced filtering, and responsive UI improvements would increase usability and scalability.
- **Dataset Expansion**: Incorporating supplementary information such as customer demographics, marketing campaigns, or product return reasons would allow richer and more robust analysis.
- **Cloud Deployment**: Hosting the Streamlit application on a cloud platform would enable external access, making the system more realistic and closer to a production visual tool.

Project Files and ZIP Archive Contents

A complete project package is included as a `SQL_FINAL_Project.zip` file. This archive contains all SQL scripts, datasets, Streamlit application code, images used in the report, and supporting files required to reproduce the results. Below is a summary of its directory structure and contents.

Root Directory

- `Schema.sql`
  Defines the full database schema, including tables, constraints, and data types for the Apple analytics database.

SQL Scripts

- `Basic_Queries.sql` — Contains introductory SQL queries for data exploration.
- `Advanced_queries.sql` — Includes joins, window functions, aggregations, and advanced analytics.
- `Creating_index.sql` — Index creation scripts used for query performance optimization.

Dashboard Code and Images

`Dashboard_Code&Photos/`

- `Code.py` — The full Streamlit dashboard source code, including page navigation, SQL execution, and visualizations.
- `Photos/` — Screenshot of the dashboard and the name of the the photos are below:
  - `ProofDashboard.png` Shows the proof of us having the dashboard on local host
  - `Visual_1.png`
  - `Visual_2.png`
  - `Visual_3.png`

  - Visual_4.png
  - Visual_5.png
  - Visual_6.png
  - Visual_7.png
  - Visual_8.png

Dataset Files

Data_folder/

- products.csv — Product reference data.
- category.csv — Product category mappings.
- stores.csv — Store metadata.
- sales.csv — Transactional sales data.
- warranty.csv — Warranty and repair records.

Linear Regression Module

Linear_Regression/

- Linear_regression_query.sql — SQL script to compute slope and intercept.
- LR_prediction_query.sql — SQL script to generate predicted sales values.
- SQL.ipynb — Jupyter Notebook with experiments and step-by-step SQL regression logic.

Purpose of the Archive

The ZIP archive is intended to:

- Provide all source code necessary to reproduce our analyses and results.

- Allow instructors and reviewers to inspect SQL development, regression models, and dashboard logic.
- Supply the datasets used for testing, demonstration, and evaluation.

Together, these files form a complete, reproducible, and well-organized submission package for the Apple Sales and Warranty Analytics project.

10 Acknowledgments

We would like to express our sincere gratitude to the Professor for providing an amazing quarter and the opportunity to make this project.

References

References

[1] PostgreSQL Global Development Group, *PostgreSQL Documentation*. Available: https://www.postgresql.org/docs/

[2] *Streamlit Documentation*. Available: https://docs.streamlit.io/

[3] *Plotly Express Reference*. Available: https://plotly.com/python/plotly-express/

[4] *Python Language Reference*. Available: https://www.python.org/

[5] U. Patel, "Apple Sales and Warranty Dataset," originally used for a machine learning project.

[6] IEEE, *IEEE Conference Paper Formatting Guidelines*. Available: https://www.ieee.org/

AI Use Acknowledgment

Portions of this report were supported by the use of AI tools, specifically OpenAI's ChatGPT, to assist with writing clarification, adding images, and fixing LaTeX-related errors. For the Streamlit implementation, AI was used

solely for research guidance on how to set up the framework and structure the application.