ECE 401C Final Report

Joseph Corella (23792538)

Bryn Neal (23830283)

Introduction / Task Definition

This project presents the design, implementation, and evaluation of a relational database system for a local bakery. The primary goal is to improve operational efficiency and support analytic tasks such as understanding consumer purchasing behavior through methods inspired by market basket analysis. The database system models several core aspects of bakery operations, including sales transactions, employee scheduling, and customer activity. The success of our system is determined by the efficiency of queries, the consistency of table data, and the insights that can be made by using multiple queries in tandem.

Throughout the development process, our main focus narrowed in on modeling sales behaviour and operational behavior. Some performance metrics explored include items frequently sold together, determining products that generate the most revenue, and when customers buy items most frequently. Overall, our project successfully models bakery sales and employee scheduling within a normalized relational framework. However, we recognize that the current system does not fully address inventory management and tracking product size, which represent opportunities for future expansion.

Database

The database developed for this project is primarily based on a public dataset from Kaggle, which contains over 20,000 bakery sales transactions recorded over the span of approximately one year. The dataset provides the date and time of each transaction, a transaction ID, and the name of the item purchased. While this information allowed us to construct a foundational sales table and product_info table, the dataset lacked several dimensions necessary for a fully functional operational model, such as employee information, customer demographics, and inventory attributes.

To integrate the public dataset into our system smoothly, we used external tools such as Excel to map our product IDs correctly. However, because the Kaggle dataset did not include customers, employees, or product metadata, we generated synthetic data using ChatGPT. This included approximately 200 unique customers, 15 employees, and extended product attributes such as product descriptions, nutrition facts, and price values. With this expanded dataset, we were able to design and populate additional tables and define appropriate primary keys and foreign keys.

Although using ChatGPT-generated data was suitable for this project, some limitations emerged, such as unrealistic bakery hours, employee work hours, and the price of items. These inconsistencies highlight areas where access to real-world operational records would improve the accuracy and reliability of the model.

Before fully implementing the final database, our initial tables and relational schema were significantly simpler. As the project progressed, we refined the schema, added new relational structures, and normalized the data by transferring independent functional dependencies into separate tables. The original and final ER diagrams can be seen below.

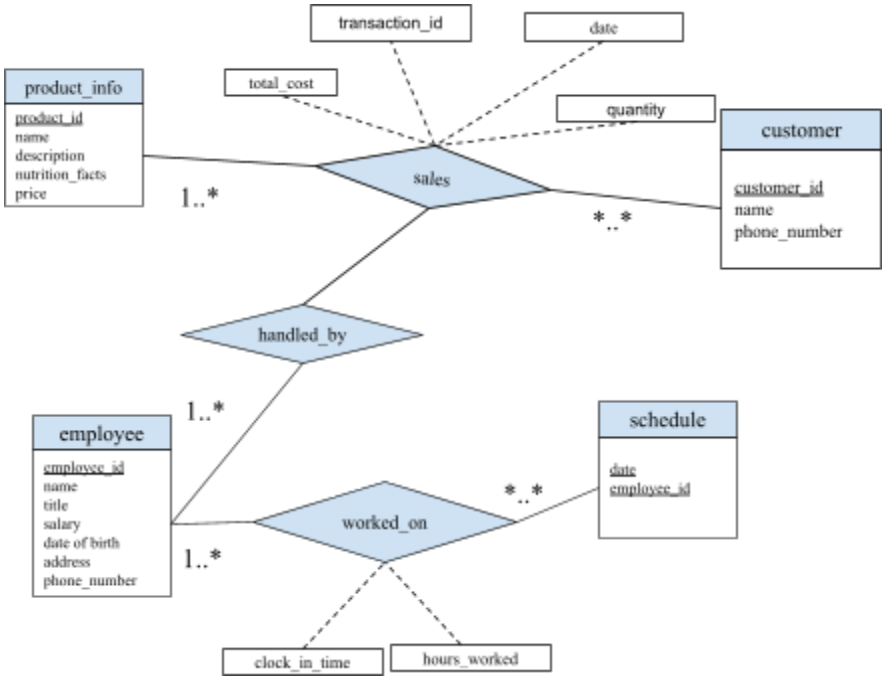

Figure 1. Initial ER Diagram

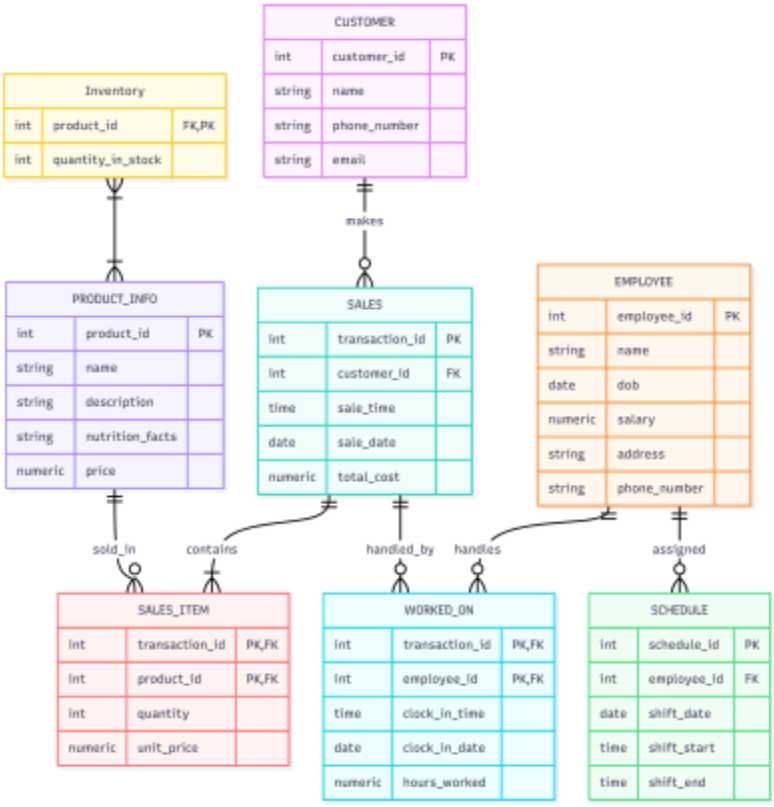

Figure 2. Final Bakery ER Diagram generated using Mermaid Chart

Infrastructure

Our initial implementation was represented in DB Browser for SQLite. However, as the project progressed, we switched to a PostgreSQL-based engine due to SQLite's limitations, particularly with respect to large imports and multi-user collaboration. Neon is a PostgreSQL database management system that supports real-time collaboration and allows all users to work on the same database. This particularly stood out to us because Neon allowed us to work online and update our queries and tables without having to download our database locally. Although we did not use the version control system very much, Neon also supports version control, allowing us to develop without pushing to production and destroying our whole database (but in most cases, we committed most changes since these are not really sensitive cases).

Constructing the database required importing the Kaggle dataset and the synthetic data generated for customers, employees, and products. While the tables were successfully created and populated, as shown in Figure 2 above, the migration process presented several challenges. Converting SQL scripts from SQLite to PostgreSQL required updates to data types, constraint syntax, and formatting rules. Additionally, uploading load datasets directly through its interface was not supported. Thus, our sales table, which contains approximately 20,000 transactions, could not be imported in a single step.

To overcome this, we used the terminal-based psql tool to load data directly into the Neon database. This required preparing clean, properly formatted CSV files and ensuring consistent data types across all fields. To do this, we used Excel and supplementary SQL scripts.

Despite these difficulties, the migration ultimately resulted in a more powerful, scalable, and reliable system with better visualization. PostgreSQL's advanced features, such as triggers and indexing, along with Neon's collaboration tools, ultimately provided a stronger foundation for the development and analysis of the bakery database.

Approach

The approach for this project centered on designing an efficient database that accurately models the operational processes of a local bakery while supporting efficient analytical queries. One of the primary challenges we faced was the limited scope of the available public dataset. To overcome this limitation, we restructured the existing dataset and generated additional tables for customers, employees, and enhanced product information, transforming a single-entity dataset into a multi-entity relational system.

Data modeling challenges also emerged during our early design phases. Initial versions of our schema were overly simplified, with some relationships defined improperly or merged into single tables. To ensure that each table captured a single conceptual entity and that all functional dependencies were properly isolated, we attempted to use normalization principles to get tables into Third Normal Form (3NF). Normalizing the tables resulted in a more coherent schema; however, our tables are not currently in 3NF, as we have been updating our schema further. To understand the impact of our design decisions, we can compare the performance of queries on the normalized schema with PostgreSQL's automatic indexing against simple baselines, including unnormalized versions of the tables and unoptimized queries from the original SQLite implementation. These comparisons demonstrate that normalization combined with PostgreSQL's default indexing resulted in noticeably faster and more reliable query execution.

In addition to structuring the schema, we considered how to optimize query performance. Although we did not implement manual indexing, the Neon PostgreSQL server automatically created B-tree indexes for primary keys and foreign keys. These system-generated indexes significantly improved the efficiency of join operations and analytical queries such as revenue summaries.

We also took transactional correctness into consideration when designing our database. To prevent invalid entries in the sales_item table, we implemented a trigger that rejects negative quantity values and ensures that each transaction represents a valid sale. This trigger enforces a critical business

rule and maintains data integrity regardless of how data is inserted into the system. We attempted to add additional triggers to the product table as well; however, these triggers are currently disabled as the schema continues to evolve. Despite this, the implemented triggers contributed to consistent behavior across transactions and improved the reliability of the dataset.

Overall, our approach combined data modeling, normalization, baseline comparison, and indexing to create a system that accurately represents bakery operations. The design addressed the challenges of the limited dataset while enabling meaningful insights into sales trends, customer behavior, and employee performance.

Features & Queries

Our system supports a variety of queries that allow us to explore different aspects of bakery performance and sales stats. The database tracks sales stats that include: time and date bought, products bought, employees who handled the sales, employee schedules, “inventory” (although incomplete), and product info. Using these tables, we created queries that include analyzing top-selling products, a 7-day moving average of revenue, and daily sales revenue. (A sample of our queries and database files is available in the appendix section of the report.)

Neon provides a great interface for analyzing the time of completion of a query, depending on the indexing techniques and operations used. Neon shows clearly what operations took what percentage of the total time, allowing for deeper analysis into our system as we test it in the error analysis.

Literature Review

The main resources we viewed at the very beginning of our project were IBM's *Transaction Management* and a conference paper, *Unveiling Consumer Behavior Patterns: A Comprehensive Market Basket Analysis for Strategic Insights,* written by Ishita Sajwan and Rajnee Tripathi.

We focused on the second paper for this project and hoped to implement ACID properties in the database later (which we unfortunately ran out of time). Consumer behavior patterns helped explain which trends to look for in our database and which queries to formulate, depending on specific variable patterns. The paper specifically focused on market-basket analysis, consumer grouping based on purchase trends, and mining/analysis techniques. From this paper, we took the idea of analyzing items together and possibly grouping customers based on purchase similarities.

For IBM's paper, it reiterated the same transaction management properties that our book went over. We'd planned to commit ACID properties and the states of insertion into our project, but unfortunately, we ran out of time. In the future, we will create states that prevent unfinished sales from being entered and destroying our data. We would implement this by adding more constraints and finding a way to insert states that prevent unfinished data from being committed.

Error Analysis

To evaluate the impact of indexing on query performance, we created duplicate versions of our tables without primary or foreign key constraints, ensuring PostgreSQL would not generate automatic indexes. This allowed us to run identical queries on both indexed and non-indexed tables and directly compare execution times using "Explain Analyze". The first benchmarked query computed the total number of hours worked by each employee. On the non-indexed tables, the query executed in 147 ms, while the indexed version completed in 104 ms. The improvement reflects more efficient join operations

when PostgreSQL is able to utilize B-tree indexes on key columns. A similar trend appeared in the analysis of product pairs frequently bought together. Without indexing, the query executed in 165 ms, whereas with indexing it executed in 107 ms. The results can be viewed in Figure 3 below. Although the performance difference is modest due to the relatively small dataset size, the results demonstrate the measurable benefits of indexing for join-heavy analytical queries.

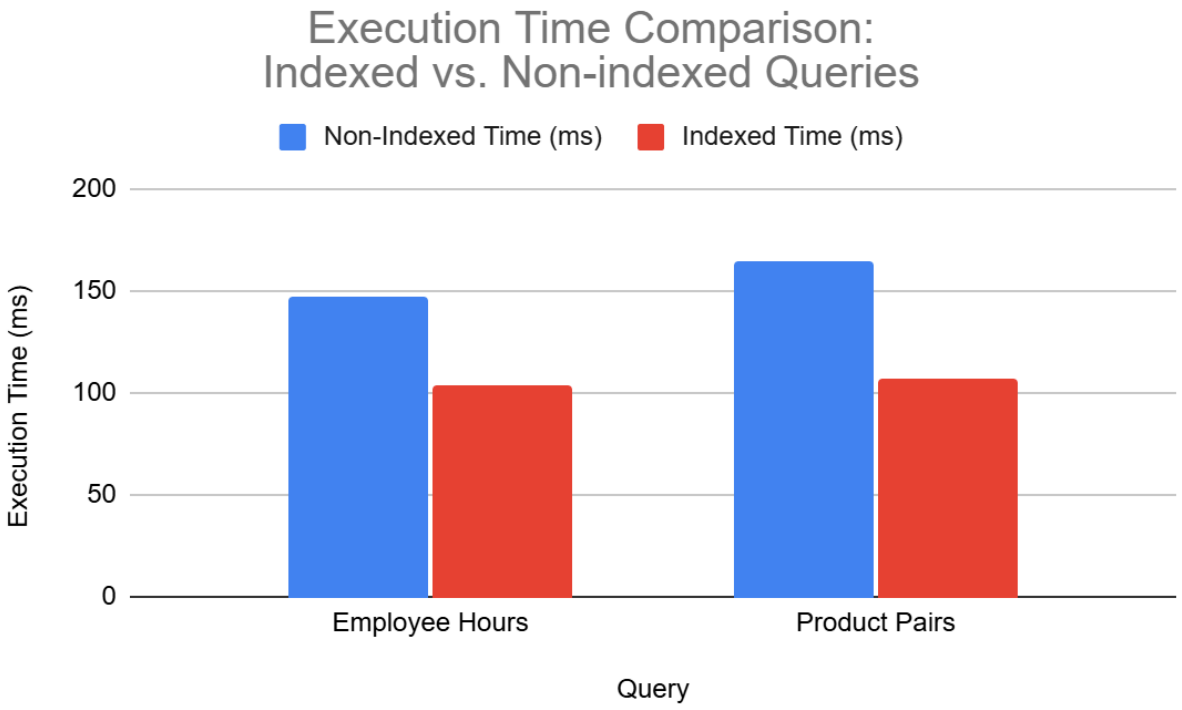

Figure 3. Execution Time Comparison for Indexed vs Non-Indexed Queries

We also compared the performance and outputs of queries run on our original SQLite implementation versus the normalized and expanded PostgreSQL database hosted on Neon. Since SQLite and PostgreSQL differ in syntax and supported features, we first converted our original queries into PostgreSQL-compatible versions. To assess differences in execution time, we ran the same analytical queries on both systems. The first query identified the best-selling item per day. In DB Browser for SQLite, using our unnormalized tables, the query executed in 51 ms and returned 159 rows. In contrast, the PostgreSQL version on Neon executed in 148 ms and returned 73 rows. The difference in row count is

attributed to schema changes during normalization and the reduced dataset size in our PostgreSQL implementation. We then tested a query that counted how many times each product was purchased over a six-month period. In SQLite, the query ran in 35 ms and returned 95 rows; in Neon, it ran in 140 ms but returned the same number of rows. Similarly, when computing a 7-day moving average of daily revenue, the SQLite version executed in 44 ms over 159 rows, while the PostgreSQL version executed in 156 ms over 73 rows. The results can be viewed in Figure 4 below:

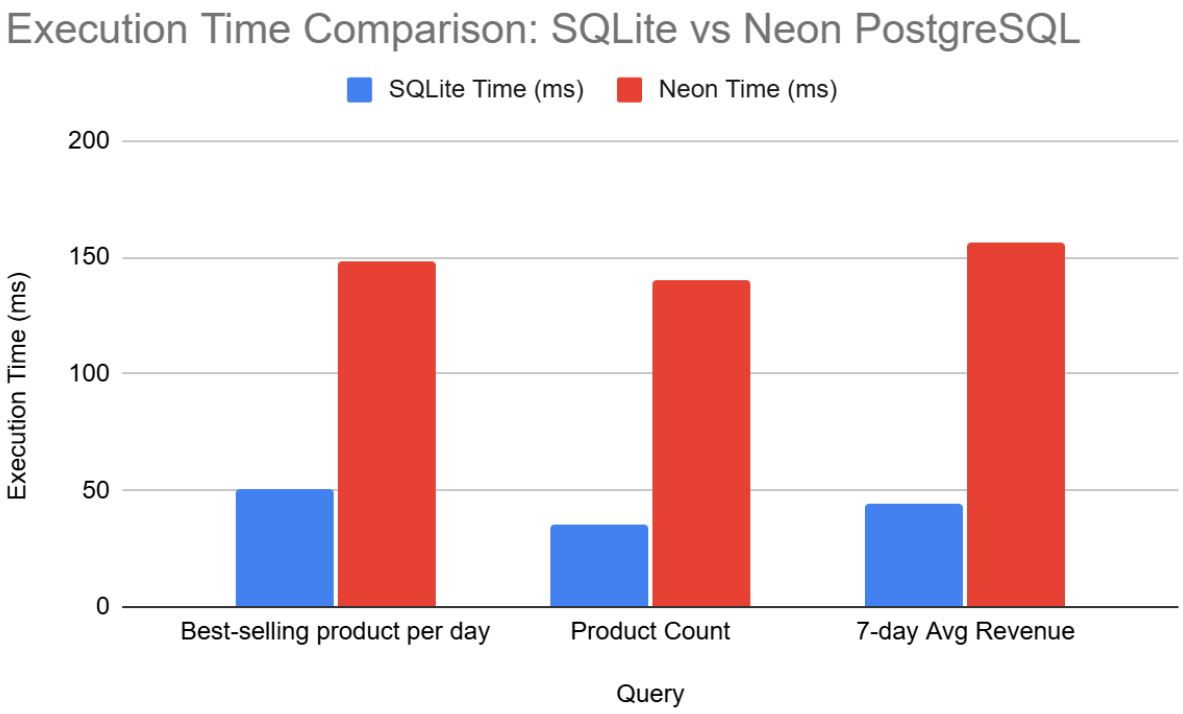

Figure 4. Execution Time Comparison for queries ran on SQLite vs Neon PostgreSQL

Across all queries, SQLite was consistently faster, but these results are expected given SQLite's smaller dataset, lack of normalization, in-memory execution model, and simpler query planner. Neon's PostgreSQL, although slower for these small-scale tests, provides stronger indexing, concurrency support, normalization, and data integrity which are features essential for scalability beyond this project.

Direction

There is strong potential for future enhancement within this project. With Neon's flexible development interface, there are numerous opportunities to improve performance and evolve the database structure over time. One area of exploration is schema optimization: additional testing could be conducted to determine whether further normalization would reduce redundancy and improve efficiency, or whether partial denormalization might better support high-frequency operations. Similarly, reevaluating indexing strategies (such as experimenting with composite or hash indexes) may lead to faster lookups and more efficient query execution.

Another promising direction involves modernizing the inventory system. Implementing real-time triggers that automatically adjust stock levels when sales records are inserted would reduce manual input and increase overall reliability. These triggers could also handle data routing internally, splitting new entries into their appropriate tables without the need to preprocess information through Excel, thereby minimizing human error and streamlining workflow.

Future development could also focus on improving automation and maintainability through stored procedures and functions, enhancing security with role-based access controls, and increasing scalability with caching as the system grows. Implementing audit logging and performance monitoring tools would provide long-term insight into database usage, helping administrators track changes, locate bottlenecks, and plan for expansion. Over time, the project could even be extended with a dedicated web interface or API layer, reducing reliance on direct SQL interaction and making the system more user-friendly.

By continuing to refine the architecture and build automated processes, this database has the potential to grow into a highly scalable, efficient, and production-ready solution. There is potential for future development on this project. Due to the flexibility and development interface of Neon, there is room for improvement in query performance and table normalization. Examples include testing if

normalization would slow down the database or enhance it, or potentially changing the indexing to a different format to help enhance queries.

Appendix

Appendix A - Queries

1.) **Top 10 Product Pairs Frequently Sold Together**

```
SELECT
   p1.name AS product_a,
   p2.name AS product_b,
   COUNT(*) AS count_together
FROM sales_item a
JOIN sales_item b
   ON a.sale_id = b.sale_id
  AND a.product_id < b.product_id
JOIN product_info p1
   ON p1.product_id = a.product_id
JOIN product_info p2
   ON p2.product_id = b.product_id
GROUP BY p1.name, p2.name
ORDER BY count_together DESC
LIMIT 10;
```

2.) **Employee Total Hours Worked in Descending Order**

```
SELECT
   e.employee_id,
   e.name,
   SUM(EXTRACT(EPOCH FROM (s.shift_end - s.shift_start)) / 3600) AS total_hours
FROM schedule s
JOIN employee e
   ON e.employee_id = s.employee_id
GROUP BY e.employee_id, e.name
ORDER BY total_hours DESC;
```

3.) **List All Products with Price and Stock**

```
SELECT p.product_id, p.name, p.price, i.quantity_in_stock
FROM product_info p
LEFT JOIN inventory i ON p.product_id = i.product_id
ORDER BY p.product_id;
```

4.) **Sales That Contain More Than 5 Items**

```
SELECT
```

```
    s.sale_id,
    COUNT(*) AS num_items
FROM sales s
JOIN sales_item si ON s.sale_id = si.sale_id
GROUP BY s.sale_id
HAVING COUNT(*) > 5;
```

5.) Best Selling Item Per Day

```
SELECT DISTINCT ON (s.sale_date)
    s.sale_date,
    p.name AS top_item,
    COUNT(*) AS count
FROM sales s
JOIN sales_item si ON s.sale_id = si.sale_id
JOIN product_info p ON p.product_id = si.product_id
GROUP BY s.sale_date, p.name
ORDER BY s.sale_date, count DESC;
```

Appendix B - Neon Database

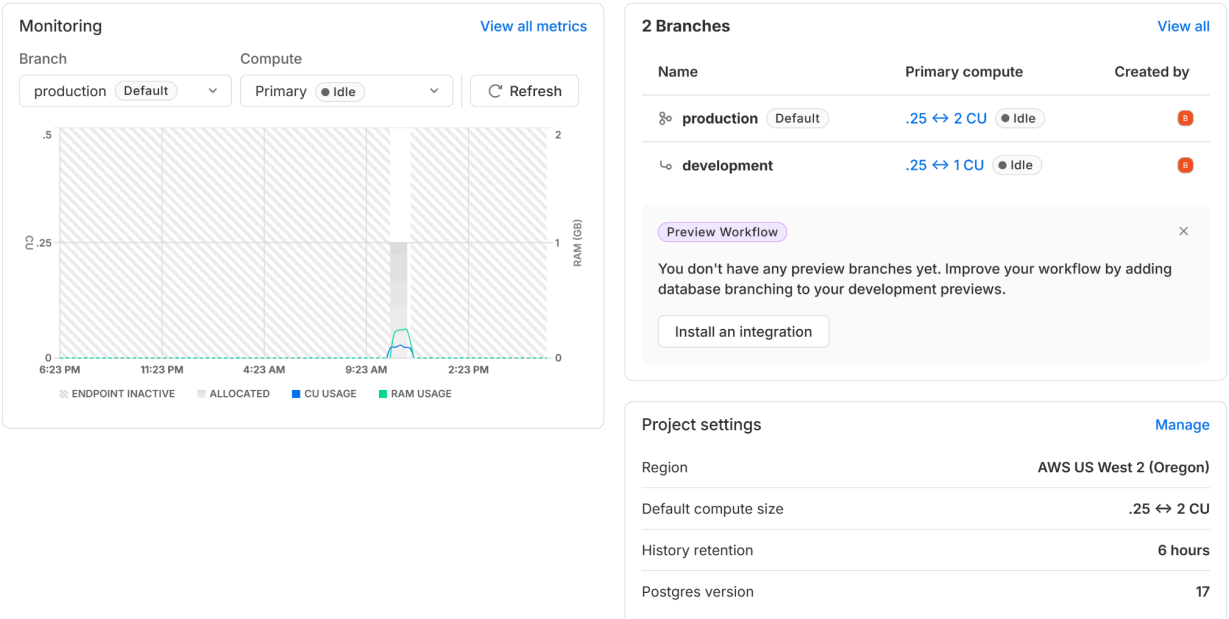

Figure B1: Neon database dashboard displaying system metrics and branch configuration.

sale_id serial	sale_date date	customer_id integer	sale_time time
1	2016-10-30	165	09:58:11
2	2016-10-30	21	10:05:34
3	2016-10-30	41	10:05:34
4	2016-10-30	135	10:07:57
5	2016-10-30	81	10:07:57
6	2016-10-30	114	10:07:57
7	2016-10-30	42	10:08:41
8	2016-10-30	60	10:13:03
9	2016-10-30	157	10:13:03
10	2016-10-30	13	10:13:03
11	2016-10-30	140	10:16:55
12	2016-10-30	189	10:16:55
13	2016-10-30	162	10:16:55
14	2016-10-30	128	10:19:12
15	2016-10-30	77	10:19:12
16	2016-10-30	121	10:19:12
17	2016-10-30	156	10:19:12
18	2016-10-30	198	10:20:51
19	2016-10-30	182	10:20:51
20	2016-10-30	87	10:21:59

Figure B2: Sample view of the *sales* table in Neon PostgreSQL displaying sale_id, sale_date, customer_id, and sale_time for 20 transactions.

Figure B3: Schema view of the *sales* table including column definitions, primary and foreign key constraints, and indexes

Appendix C - Citations

Sajwan, I., & Tripathi, R. (2024, June). *Unveiling consumer behavior patterns: A comprehensive market basket analysis for strategic insights*. In *2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT)*. https://doi.org/10.1109/CCICT62777.2024.00067

Schneider, J., & Smalley, I. (n.d.). *What is transaction management?* IBM Think. https://www.ibm.com/think/topics/transaction-management

Appendix D - Dataset

Public Bakery Data - https://www.kaggle.com/datasets/akashdeepkuila/bakery