SQL-like DataFrame from Scratch: *Under the Hood*

Project Final Report - submitted

by

Rizwan Ahasan Pathan
USC ID: 6909 753 128

Under the Supervision

of

Wensheng Wu

Department of Computer Science
Viterbi School of Engineering
University of Southern California (USC)

December, 2025

1 Project Overview

The motivation behind this project was to deeply understand how databases operate at the fundamental level by replicating core SQL logic from scratch in Python. Rather than relying on high-level libraries like `pandas`, `csv`, or `json`, the goal was to manually implement key data operations—such as selection, filtering, joining, and aggregation—using only native data structures. This approach offered a hands-on learning opportunity to explore how real-world SQL engines function internally.

At its core, the project implements a fully functional, in-memory SQL-like query engine. It supports essential operations including:

- **Projection (SELECT)**
- **Filtering (WHERE)** with compound conditions
- **Grouping (GROUP BY)** and **Aggregation** (`COUNT`, `AVG`, `MIN`, `MAX`)
- **Joins** (`INNER`, `LEFT`)
- **Sorting (ORDER BY)** with customizable direction
- **Indexing and Primary Key Support** to improve performance and enforce uniqueness

To make the system accessible and interactive, a web-based frontend was developed using Streamlit. Users can construct queries visually through dropdowns and input fields, or use an advanced mode to write queries in code via the `select_query()` function. The application is deployed on Streamlit Cloud and publicly available at: `https://onlypathan.streamlit.app`

The final system bridges backend data logic with frontend usability, enabling fast, intuitive data exploration on real-world datasets—all within a fully self-built SQL-like framework.

2 Algorithms for Loading, Parsing, and Operations

2.1 CSV Parsing (read_csv_custom)

A custom parser implemented to load and process CSV files, mimicking the behavior of `pandas.read_csv()` while avoiding any CSV/JSON/pandas libraries. The parsing process includes the following steps:

- Manually tokenizes file content (no `csv.reader`) to build fields/rows, handling quotes and delimiters.
- Uses the first row as trimmed headers to define the schema.
- Infers per-cell types (`int` → `float` → `str`); empty cells become `None`.
- Yields row dictionaries (header → value) and pads/truncates rows to match the header length.

2.2 Internal Table Structure

A custom class named `MyCustomDB` was developed to function as a simplified in-memory DataFrame. Internally, it stores data as a dictionary of columns:

- {*column_name*: [*values*]}
- Supports row access, column selection, and column-wise operations

The system is designed using loose coupling principles, meaning that each major operation—such as filtering, joining, or grouping—is implemented as an independent module. As a result, each function can be executed, modified, or even removed without affecting the rest of the system's integrity. This promotes flexibility, testability, and maintainability.

To handle table creation and data ingestion, a separate `DataLoader` class was implemented. It is responsible for:

- Creating tables with defined primary keys and indexed columns

- Parsing and loading all CSV files into memory using the custom parser
- Keeping data loading logic modular and decoupled from query execution logic

This architectural choice improves code modularity, enhances reusability, and enables scalability for integrating additional datasets in the future.

2.3 SQL Operation Functions

The main function used to execute queries is `select_query()`, which accepts parameters corresponding to standard SQL operations. This modular design allows each operation to be included or omitted independently, enabling flexible and dynamic query execution.

Table 1: Supported SQL-like operations in `select_query()`

SQL Operation	Parameter	Description
SELECT	`columns=[...]`	Specifies which columns to return in the result.
FROM	`from_table="..."`	The base table for the query.
WHERE	`where=[[(col, op, val), ...]]`	Applies filtering conditions; supports nested logic using AND / OR.
JOIN	`joins=[(table, (key1, key2), type)]`	Joins another table using specified keys and join type (e.g., `inner`, `left`).
GROUP BY	`group_by="..."`	Groups rows for aggregation based on the selected column.
AGGREGATE	`agg_fn="..."`, `agg_col="..."`	Computes aggregate metrics such as `avg`, `count`, `min`, and `max`.
ORDER BY	`order_by=[...]`, `descending=[...]`	Sorts output by one or more columns with control over sort direction.

Primary Key and Indexing Support: In addition to the operations above, the system supports defining primary keys and column-level indexes during table creation. Primary keys ensure data uniqueness and significantly optimize join operations by enabling hash-based lookups, reducing time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Indexing on frequently queried columns enables faster filtering and data access, with potential complexity reductions to $\mathcal{O}(1)$ in ideal cases using in-memory hash maps.

These features are inspired by foundational principles of relational databases and enhance both performance and scalability. The engine processes all query instructions in a modular and sequential manner, making it adaptable to a wide range of query types and user needs.

3 Datasets and Application

3.1 Datasets Used

The application operates on four structured CSV datasets, each representing key aspects of restaurant operations and neighborhood demographics in Los Angeles. The datasets were originally scraped from public web sources and compiled for academic purposes. They capture a one-year snapshot of real-world data and support a wide range of query and analysis tasks.

Table 2: Summary of datasets used in the application

File Name	Description
`restaurant_info.csv`	Contains restaurant metadata such as name, category, ZIP code, and review count.
`inspection_info.csv`	Includes health inspection scores and grading information for each restaurant.
`demographics_info.csv`	Provides demographic indicators by ZIP code, including population, income, and employment data.
`zip_code.csv`	Contains ZIP code classifications and relevant metadata for location mapping.

All datasets are stored locally in CSV format and loaded into memory using a custom-built parser. This enables efficient execution of SQL-like operations—within the application's interactive query interface.

3.2 Application Overview

The application is a custom-built, interactive query tool developed using Streamlit, which provides a modern and responsive web interface for users to perform SQL-like data analysis on structured datasets—without writing actual SQL or Python code.

Built on Custom Backend Logic

The application is powered by a core function called `select_query()`, which supports the following key SQL-like operations:

- Table selection and joining
- Complex filtering with logical conditions (AND/OR)
- Grouping and aggregation (e.g., average scores, counts)
- Projection (column selection)
- Sorting (`ORDER BY`)

All operations are handled by a custom-built query engine named `MyCustomDB`, designed to operate entirely on in-memory data structures parsed from CSV files. This backend avoids the use of restricted libraries such as `pandas`, `json`, or `csv`, and is implemented using native Python data structures.

Streamlit-Based Frontend

The user interface, built using Streamlit, is structured into multiple modular sections that allow users to build and execute queries in a step-by-step manner. The design focuses on usability, enabling both technical and non-technical users to interact with complex datasets visually.

Figure 1: Interactive SQL-like query builder interface - Normal Query Mode

Table Selection & Join Configuration

- Users specify the base table (`from_table`) and optionally join another table.
- Joins are configured by:
  - Selecting the second table
  - Specifying join keys (e.g., `Restaurant_Info_ID` ↔ `F_Restaurant_Info_ID`)
  - Choosing join type (inner, left)
- This is entered through labeled input fields.

WHERE Filter Section

- Users define row-level filtering conditions.
- Multiple conditions can be combined using `AND` or `OR` logic.
- Inputs include:
  - Column name
  - Operator (=, >, <, etc.)
  - Value
- Logical connectors are supported in advanced queries.

GROUP BY and AGGREGATION

- Users can group data by any column (e.g., ZIP code, Category).
- Aggregation functions supported:
  - AVG, COUNT, MAX, MIN
- Aggregated columns (e.g., `inspection_info.Score`) are specified alongside the function (e.g., `avg`).

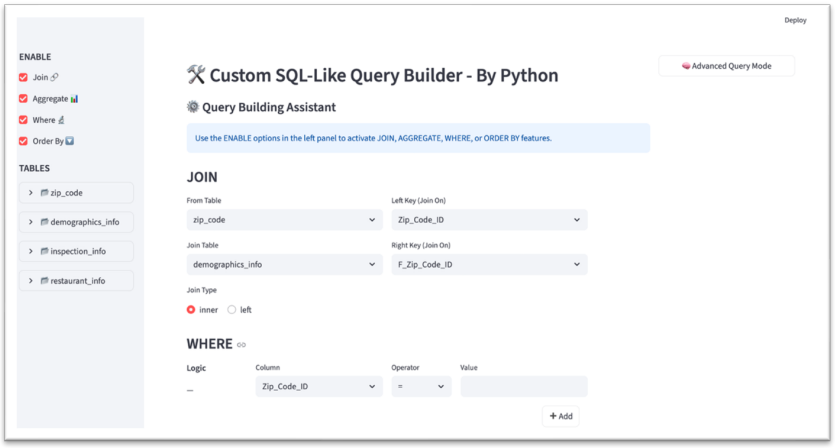

Figure 2: Join and WHERE clause configuration in the query builder interface

SELECT / Projection

- Users choose which columns to include in the final result table.
- The selected columns may include original, joined, or aggregated fields (e.g., `restaurant_info.Categories`, `avg_Score`).

ORDER BY

- Results can be sorted by one or more columns.
- Users control sort direction (ascending, descending) per column.
- This helps highlight high- or low-performing records (e.g., top ZIPs by average inspection score).

Advanced Query Support

For users with technical knowledge, the application also supports an advanced code entry mode, where they can input raw `select_query()` calls with full parameter control. This unlocks:

- Nested filters

- Custom aggregation logic
- Programmatic query chaining

Figure 3: Advanced query editor allowing full parameter-based `select_query()` input

This mirrors complex SQL queries — such as filtering, joining, grouping, and ordering — all without using SQL syntax.

Output and Accessibility

Query results are displayed in a sortable, scrollable table directly within the browser, allowing users to interactively explore and interpret aggregated insights. The application requires runs entirely in the browser, ensuring smooth access for both technical and non-technical users. This streamlined deployment model supports fast, intuitive data analysis across different use cases.

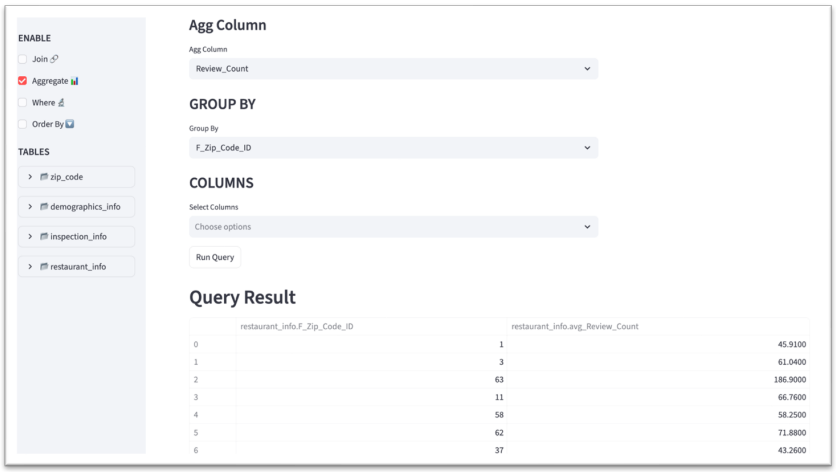

Figure 4: Average number of reviews grouped by zip code using aggregation on the Review Count column.

3.3 How to Run

Environment: Python 3.10+ (tested locally).

1. **Place all datasets** in the `data/` directory. The engine can analyze any CSV files that follow the required column names and data types for each dataset. Example file layout:

- `data/restaurant_info.csv`
- `data/inspection_info.csv`
- `data/demographics_info.csv`
- `data/zip_code.csv`

2. **Install dependencies**
   If `streamlit` is not installed, install it directly:

   ```
   pip install streamlit
   ```

3. **Run the UI (Streamlit)**

   ```
   streamlit run index.py
   ```

   This opens the browser interface. Load the CSVs, choose operations (filters, joins, group/aggregate, order), and execute queries.

4 Learning Experiences

This project offered a rich, hands-on learning opportunity that deepened my understanding of both theoretical and practical aspects of data systems development. By building a SQL-like query engine from scratch—without relying on libraries like pandas, csv, or json—I gained a deeper appreciation for how data operations work at a lower level.

One of the key learning outcomes was developing an understanding of how SQL query logic is processed internally. Reconstructing operations such as `SELECT`, `WHERE`, `JOIN`, `GROUP BY`, and aggregation allowed me to simulate the control flow and modular processing that real-world query engines perform. I had to think critically about row-based vs. column-based access, memory efficiency, and computational trade-offs in every operation.

I also learned how DataFrame-like structures can be built and manipulated purely in Python. Designing the `MyCustomDB` class helped me realize how internal data representation affects performance. Implementing indexing and primary key constraints gave me practical experience in optimizing search and join operations, and understanding the trade-offs of different data access patterns.

From a developer perspective, this project strengthened my skills in code modularization and abstraction. By adhering to loose coupling principles, I was able to create independent functional blocks (e.g., filtering, grouping, joining), making the system extensible and easier to debug or enhance later. This architecture not only made the system more maintainable but also laid a foundation for future scalability.

On the frontend side, building the Streamlit-based interface helped me understand how to design user-friendly, step-by-step workflows that abstract complex backend logic into a visual experience. I learned how to turn raw backend functionality into something accessible and intuitive for end users, including both technical users (via advanced query input) and non-technical users (via GUI-driven queries).

Deployment via Streamlit Cloud taught me how to expose backend applications to real users with minimal infrastructure. I learned to manage environment dependencies, test performance under different workloads, and ensure compatibility across platforms.

In addition, I enhanced my skills in:

- Debugging complex data pipelines
- Performance profiling and memory-efficient design
- Maintaining version control using GitHub for clear progress tracking

Overall, this project pushed me to combine backend algorithmic thinking with frontend design, resulting in a well-rounded experience that mirrors what real-world data engineers and full-stack developers encounter. It was not only technically challenging but also personally rewarding.

5 Contribution Summary

As a solo participant, I was fully responsible for:

- Designing and implementing the entire backend engine
- Writing all functions for parsing, filtering, grouping, sorting, and joining
- Building the Streamlit-based frontend interface
- Integrating frontend and backend
- Hosting and deploying the app
- Writing all documentation and preparing demo materials

I also maintained version control and commit history via GitHub to track progress over time.

6 Conclusion

The project demonstrates a practical and functional system for interactive data analysis, integrating core query operations with a user-accessible web interface. It effectively bridges backend logic with frontend usability, allowing users to construct and run complex queries either visually or through advanced input.

Looking ahead, several enhancements are planned to extend functionality and improve user experience. These include enabling users to upload and explore their own CSV files at runtime, supporting query saving and result exporting, and adding visual analytics such as charts to better summarize grouped data. To support larger datasets, chunked file reading may be introduced to optimize memory usage during data loading.

Overall, the project satisfies requirements by incorporating a deployable GUI, modular backend logic, and a foundation that supports future scalability. It reflects both technical and practical understanding of relational query processing and interactive data exploration.

References

[1] Streamlit Inc. *Streamlit Documentation*. Retrieved from `https://docs.streamlit.io`, 2023.

[2] Python Software Foundation. *Python Official Documentation*. Retrieved from `https://docs.python.org/3/`, 2023.

[3] SQLite Consortium. *SQLite Query Planning and Indexing*. `https://sqlite.org/queryplanner.html`, 2023.

[4] PostgreSQL Global Development Group. *PostgreSQL Index Types*. `https://www.postgresql.org/docs/current/indexes.html`, 2023.

[5] R. A. Pathan. *SQL-LikeEngine-ByPython (GitHub Repository)*. `https://github.com/onlypathan/SQL-LikeEngine-ByPython`, 2025.

[6] R. A. Pathan. *SQL Engine Web Application (Streamlit)*. `https://onlypathan.streamlit.app/`, 2025.