Name: Likhit Jha
USC ID : 4473020128
Course: DSCI 551

Link:

https://drive.google.com/drive/folders/1EbGjIe8bHAp5iRRou9o7mljl7EDQamXU?usp=sharing

Introduction

This report summarizes an in-depth implementation of Python CSV data manipulation methods. This document provides a thorough walkthrough of a number of actions, such as grouping, aggregating, filtering, and running SQL-like queries on sizable CSV datasets. This project's main goal was to create a reliable Python script that could handle large CSV files quickly and effectively while carrying out common SQL query operations. Developing scalable, efficient, and modular code to carry out various data manipulation tasks was the aim.

• **Chunk-Based Processing:** Recognizing the importance of chunk-based processing and how it helps with effective memory management when working with large CSV files. Memory Size is assumed to be 3000 rows and Chunks are of 1000 rows

• **Custom Query Execution:** By including functions that can parse and run SQL-like queries on CSV data, users will be able to manipulate data in a variety of ways.

• **Modular code design:** Creating code modules with readability, maintainability, and scalability in mind so that functions can be reused for a variety of data manipulation tasks.

Crime Dataset: 7 columns and 10,000 rows

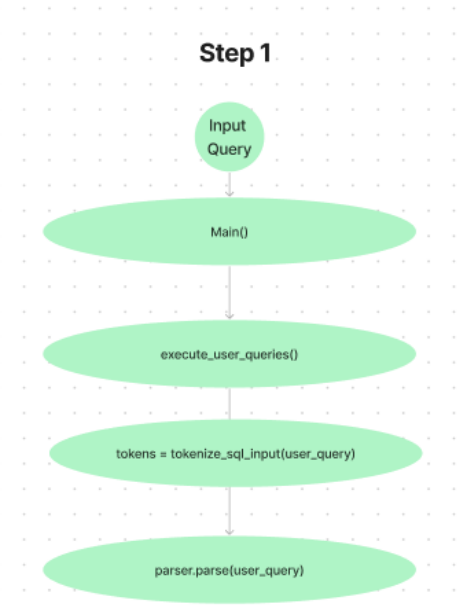
LA_Crime_Dataset 2022: The project was tested on this data as well. 9 Columns and 10,000 rows Memory Size is assumed to be 3000 rows and Chunks are of 1000 rows

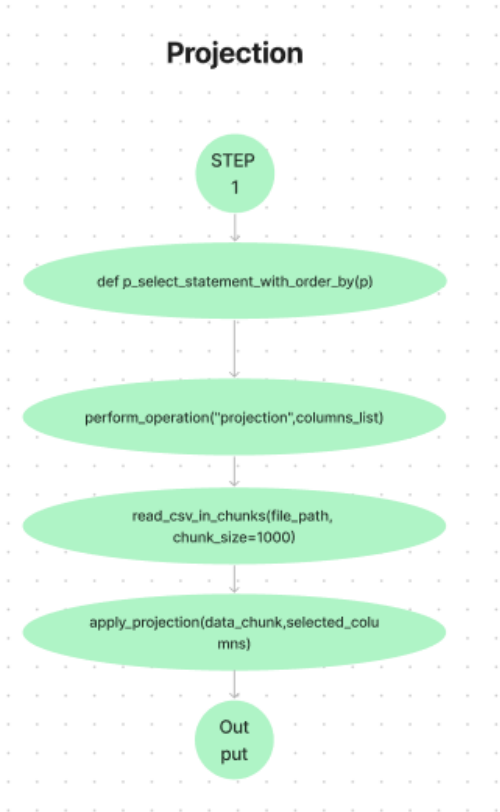
Planned Implementation (From Project Proposal)

- Since the assumption is that the main memory is small, and the size of the data is more than 5 times of the main memory. I plan to use a technique that loads batches of data in the memory for data modification.
- Low Language coding will be done using Python Language
- Data Model: Since the assumption is that the main memory is small, and the size of the data is large. I will be using a technique that loads batches of data in the memory for performing operations.
- Making Algorithms: Initially I will query the data using python for developing the algorithms required for joining, sorting, grouping etc.
- Testing Phase1: Testing will be done using testcases and other datasets.
- Custom Query Language Parser: Once I am done, with the implementations of all operations, I will switch to Custom Query Language. The Query Language will require a parser which will be able to convert the my custom query language
- Testing Phase2: Testing will be done to check the parser
- Project Testing: Will be testing the whole project with the custom query language

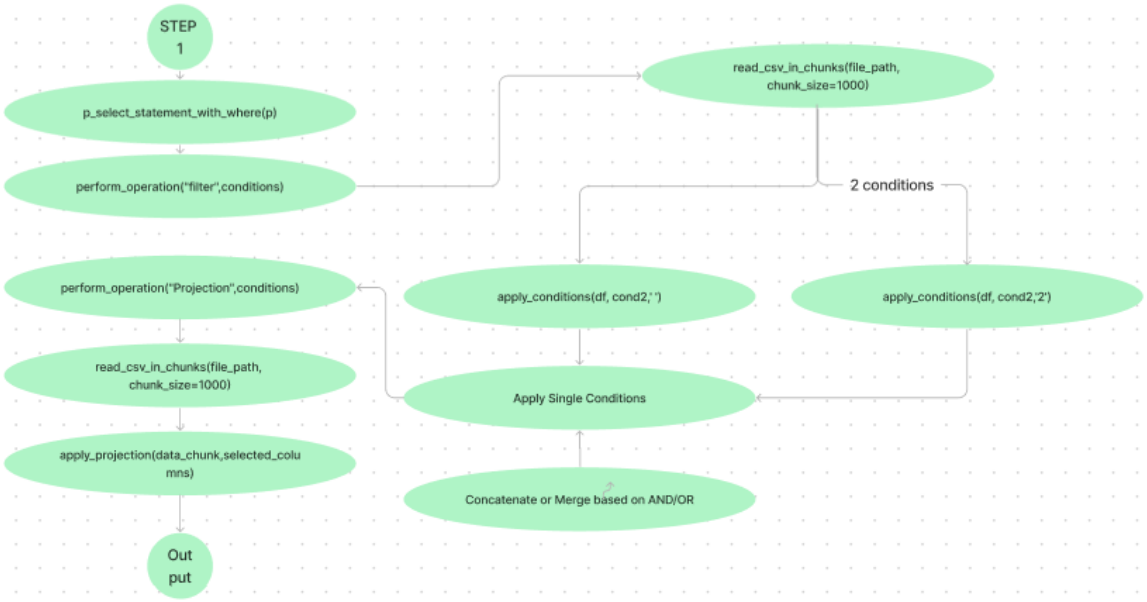
Architectural Design

The green flowchart explains the function calls that are made sequentially when an input is given. Different inputs trigger different function calls

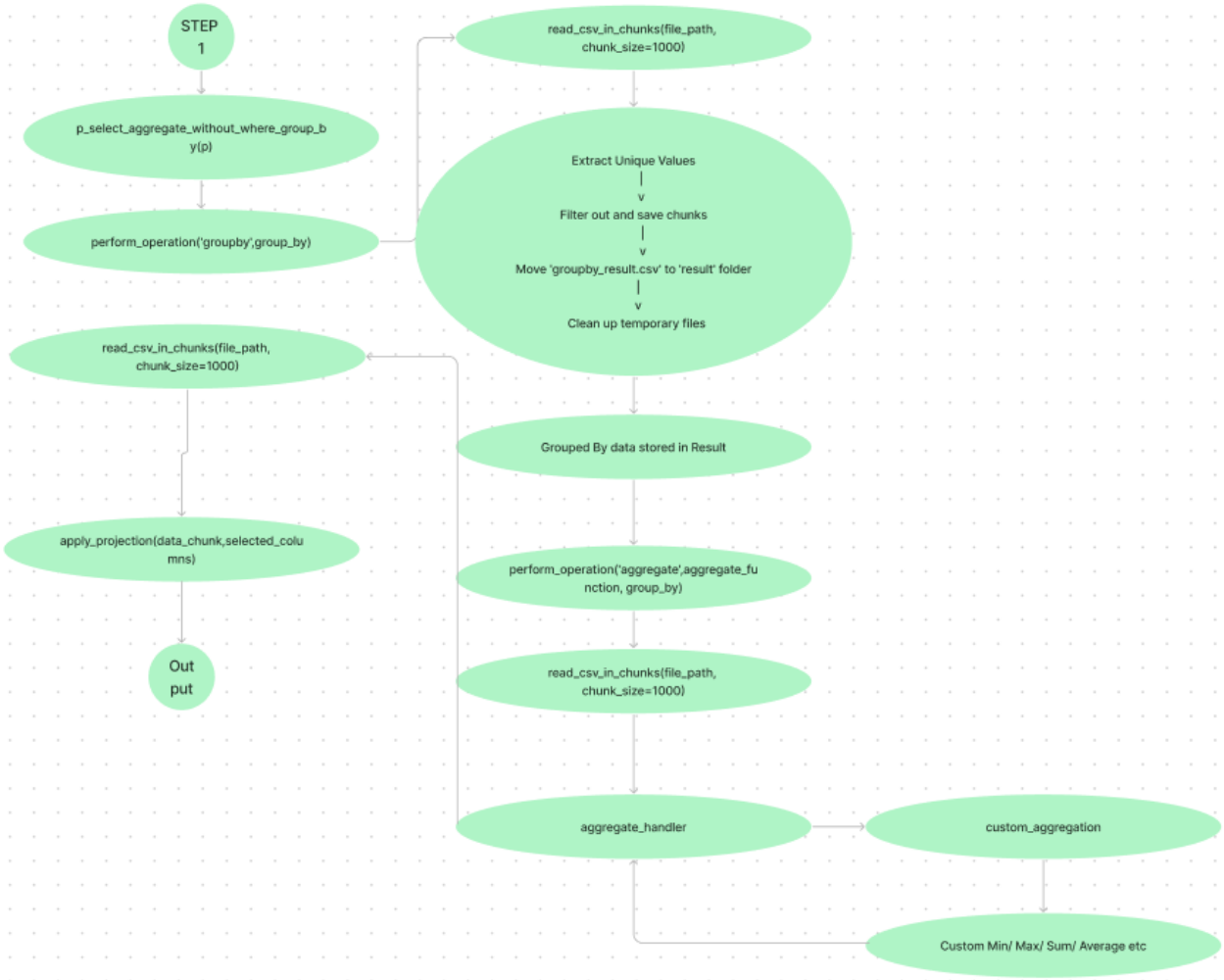


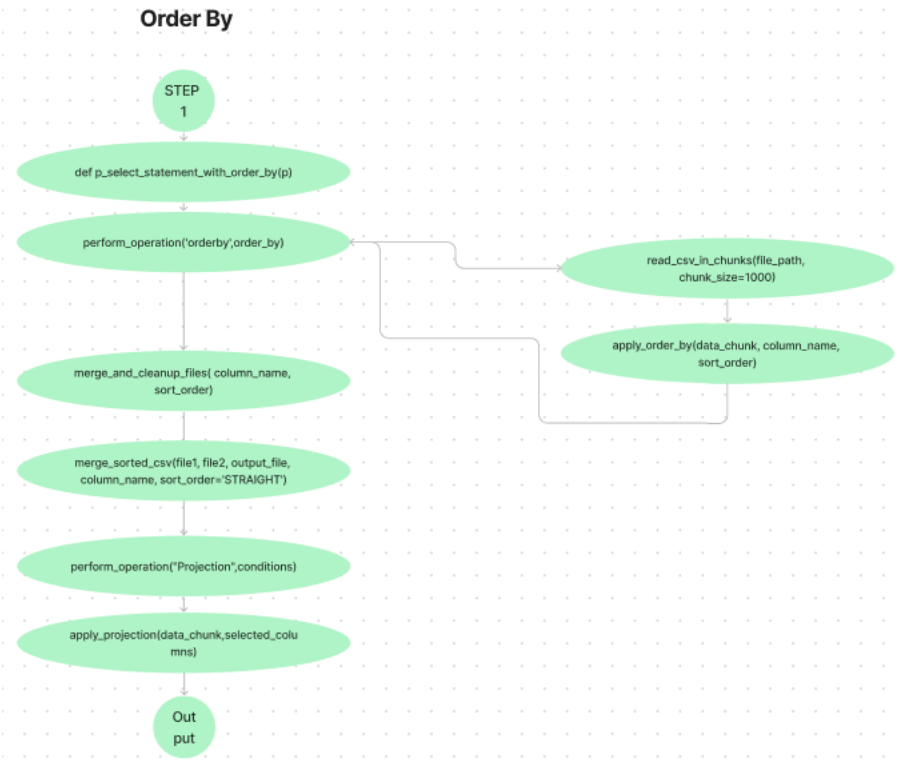


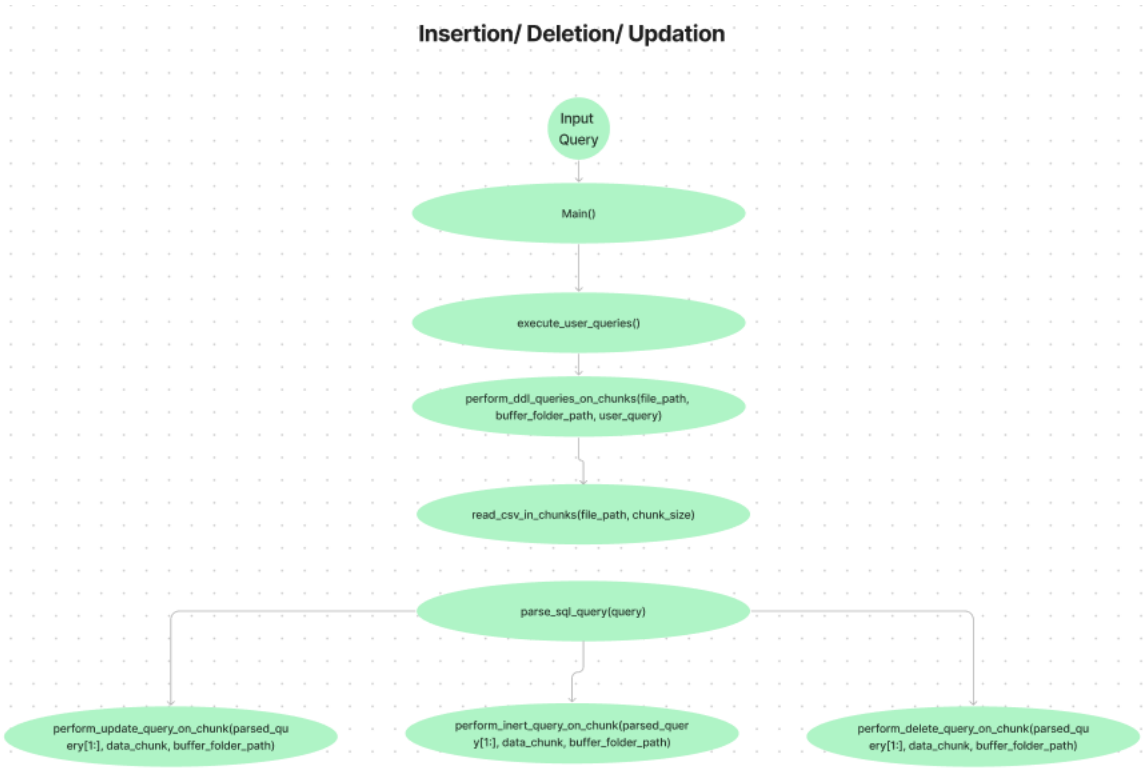
Filtering



Group By and Aggregatrion



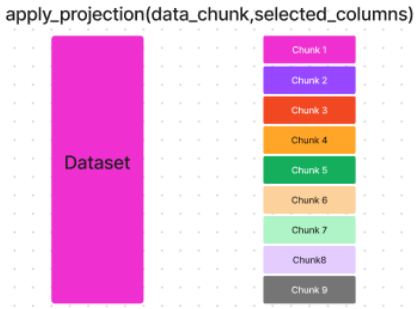




The diagrams bellow explain how the data is treated in each of the operations

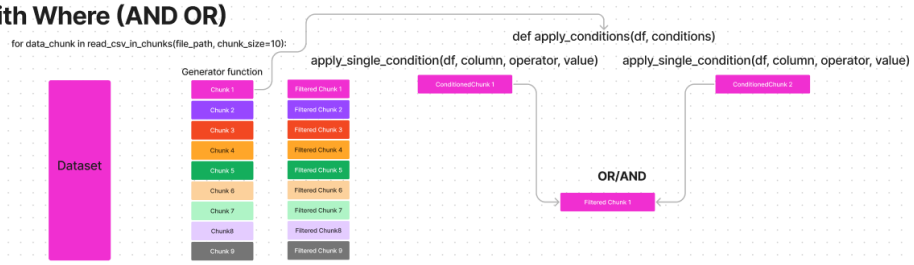
Projection

Happens in iteration
Single Conditon: Direct method on chunks
OR case: Conditon1 applied on original chunk file and Condition 2 applied on orginal chunk file, 2 chunk files are saved, then unique rows are appended from the 2 chunk files in the result csv.
AND case: Condition1 applied on 1 chunk file, then 2nd condition2 applied on that same file, end file is appended



Basic Selection with Where (AND OR)

Happens in iteration
Single Conditon: Direct method on chunks
OR case: Conditon1 applied on original chunk file and Condition 2 applied on orginal chunk file, 2 chunk files are saved, then unique rows are appended from the 2 chunk files in the result csv.
AND case: Condition1 applied on 1 chunk file, then 2nd condition2 applied on that same file, end file is appended

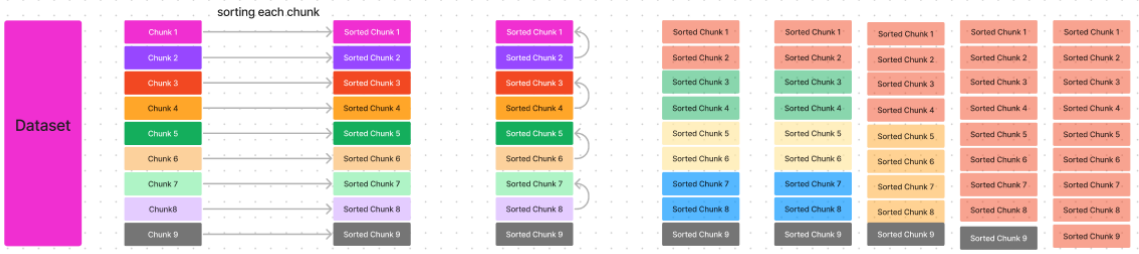


Basic Sorting

for data_chunk in read_csv_in_chunks(file_path, chunk_size=10):
apply_order_by(data_chunk,selected_columns)

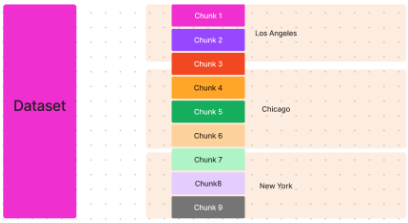
Load the first chunk in folder.,
Sort chunk1 and chunk2.
then store chunk 1 and chunk 2 in sorted folder

Take first index of sorted_chunk3
compare with last index of chunk 1
if satisfied then move to next chunk
else if not satisfied
sort chunk1 and chunk2
repeat the above process



GROUP BY

1. **Create a list of unique value for the grouped column**
2. **Now for each chunk we check for the unique value matches cell if it does we append that row to group_by_chunk**
3. **Ultimately the the new csv will have all rows in order of group by a column**



Aggregate

Aggregate handler

1. If group_by is applied
2. initializes the group_val = first val
   a. for each chunk:
   b. if group val ==current group val:
      i. performs aggregattion saves result in aggregated_value
   c. if group val !=current group val
      i. stores aggregate_dict[current_group_value] = {f"{aggregate_column}_{agg_func.__name__}": aggregated_value}
      ii. group val = current val

Dataset

for func, col in aggregate_functions:

Insertion, Updation and Deletion

perform_sql_queries_on_chunks(file_path, buffer_folder_path, query, chunk_size=10)

Insert

perform_insert_query_on_chunk(parsed_query, data_chunk, buffer_folder_path)

Insertion is perormed on the last chunk,

Update

perform_update_query_on_chunk(parsed_query, data_chunk, buffer_folder_path):

Updatetion is performed on each chunk and stored in a buffer

Delete

perform_delete_query_on_chunk(parsed_query, data_chunk, buffer_folder_path)

Deletion is performed on each chunk where possible

Implementation

Parsing Custom Query Language

Token Definitions:

- **tokens**: Describes the different Own Query Language tokens. These tokens stand in for many SQL query components, including operators (EQUALS, NOT_EQUALS, GT, LT, etc.), functions (UPPER, LOWER, SUM, COUNT, etc.), identifiers (IDENTIFIER,COLUMN_NAME,TABLE_NAME),and keywords (PRESENT, JOIN, UPDATE, etc.).

Lexer Rules:

• **t_IDENTIFIER, t_NUMBER, and t_STRING**: Regular expressions that specify how the SQL-like

language handles identifiers, numbers, and strings.

• **Reserved words and functions:** These terms are identified and mapped to their appropriate

types using definitions found in the reserved dictionary

Parsing Rules (using Ply):

The parser is built on parsing rules and can handle the following SQL-like statements:

• **p_statement:** The primary entry point for parsing various types of SQL statements.
• **p_update_statement:** Processes UPDATE statements, extracting table names, set values, and update conditions.
• **p_set_list, p_set_expression**: Set list and set expression rules in a UPDATE statement.
• **p_where_clause, p_condition:** WHERE condition rules for queries.
• **p_expression**: Expression parsing rules that include comparisons and values.
• **p_order_by_clause, p_aggregate_function, p_group_by_clause**: Rules for handling ORDER BY, aggregate functions (COUNT, SUM, etc.), and GROUP BY clauses in SQL queries.
• Rules for dealing with various types of SELECT statements that include WHERE conditions, GROUP BY, aggregate functions, and ORDER BY.

Functionality:

- **Lexing**: Tokenizes input SQL-like queries using the defined lexer.
- **Parsing**: Parses SQL-like queries into a structured format based on the defined parsing rules.
- **Handling Statements**: Different parsing rules handle various SQL statement structures like **SELECT**, **UPDATE**, **WHERE**, **GROUP BY**, **ORDER BY**, and aggregate functions.

Function Calls:

- The script includes function like **get_all_column_names**, **perform_operation** These functions are intended to perform specific operations on data based on the parsed SQL queries. **perform_operation** is called whenever the parser identifies an operation to be performed

Perform Operation (The driver function)

Objective: The code serves to perform a variety of data operations on CSV files, including projection, sorting, filtering, grouping, and aggregation. It aims to provide functionalities to process and transform CSV data based on user-specified operations and parameters.

Functionality Overview:

- **Data Operations Handling:**
  - The code defines a function **perform_operation** that is in charge of managing and orchestrating various data manipulation tasks based on the specified operation.
- **Operation Execution:**
  - The function reads data chunks from a specified CSV file path using **read_csv_in_chunks**.
  - It sequentially processes data chunks based on the specified operation.
- **Operations Supported:**
  - **Projection (operation == "projection"):**
    - Calls the **apply_projection** function to perform a projection operation on the data chunk based on provided parameters.
  - **Ordering (operation == "orderby"):**
    - Uses **apply_order_by** to sort the data chunk based on specified columns and sort orders.
  - **Filtering (operation == "filter"):**
    - Handles conditional filtering by applying conditions sequentially and saving intermediate results. Supports 'AND' and 'OR' conditions.
  - **Grouping (operation == "groupby"):**
    - Identifies unique values based on the specified column and creates separate CSV files for each unique value group.
  - **Aggregation (operation == "aggregate"):**
    - Executes custom aggregation functions based on specified aggregate functions and group-by columns.
- **Result Management:**
  - Manages the results generated by each operation:
  - Removes all files from the result folder (delete_all_files).
  - Places the resulting CSV files in the result folder.
  - If a group-by operation is performed, it optionally adds group names to the aggregated data.

Projection

Objective: The apply_projection function aims to present specific columns from a CSV file chunk in a structured, readable format. It organizes the data into a table-like display to facilitate easy interpretation and analysis.

Function Overview (apply_projection):

Applied on each chunk of data

1. **Column Alignment:** Adjusts column widths dynamically based on the length of column values, ensuring proper alignment and readability.

2. **Consistent Spacing:** Standardizes column names and data values, ensuring uniform spacing for better visual representation.
3. **Index Representation:** Includes an index column to identify rows, enhancing data referencing and navigation.

Query

```
Enter your SQL query (Type 'exit' to quit): PRESENT area_name, crm_cd_desc THIS la_crime_dataset
```

Output

9984	Southwest	BATTERY - SIMPLE ASSAULT
9985	Newton	ROBBERY
9986	Foothill	VANDALISM - FELONY ($400 & OVER ALL CHURCH VANDALISMS)
9987	77th Street	VEHICLE - ATTEMPT STOLEN
9988	Topanga	BURGLARY FROM VEHICLE
9989	Mission	BATTERY WITH SEXUAL CONTACT
9990	77th Street	VANDALISM - MISDEAMEANOR ($399 OR UNDER)
9991	Southwest	VANDALISM - FELONY ($400 & OVER ALL CHURCH VANDALISMS)
9992	Devonshire	VANDALISM - FELONY ($400 & OVER ALL CHURCH VANDALISMS)
9993	N Hollywood	THEFT PLAIN - PETTY ($950 & UNDER)
9994	Central	ROBBERY
9995	Devonshire	BURGLARY FROM VEHICLE
9996	Foothill	VIOLATION OF RESTRAINING ORDER
9997	Devonshire	THEFT PLAIN - PETTY ($950 & UNDER)
9998	Southwest	LETTERS LEWD - TELEPHONE CALLS LEWD
9999	Foothill	CONTEMPT OF COURT

Filtering

Objective: The apply_conditions function is designed to filter a chunk based on specified conditions. This function allows for the application of both individual conditions and compound conditions (combined using logical operators like 'AND' and 'OR') to efficiently filter data.

Function Overview:

- **Handling Empty Conditions:**
  - The function initially checks if there are no conditions specified. In this case, it returns the original DataFrame as no filtering is required.
- **Processing Conditions:**
  - conditions are organized into a list.
  - For each of the following conditions:
    - If it's a single condition (a tuple with three elements representing an operator, a column, and a value), the **apply_single_condition** function is called to filter the DataFrame.
    - If it's a compound condition (represented by the logical operators 'AND' or 'OR'):
    - Conditions are applied recursively to separate copies of the DataFrame (df_sub1 and df_sub2).
    - Combines results using a logical operator:
    - 'AND': Converts the filtered DataFrames into an inner merge.
    - 'OR': Concatenates and removes duplicates from the filtered DataFrames.
- **Dealing with Nested Conditions:**
  - The function handles nested conditions by recursively applying the **apply_conditions function** to DataFrame subsets.

Query

```
Enter your SQL query (Type 'exit' to quit): PRESENT area_name, crm_cd_desc THIS la_crime_dataset MENTIONED vict_age > 90
```

Ouput

```
Enter your SQL query (Type 'exit' to quit): PRESENT area_name, crm_cd_desc THIS la_crime_dataset MENTIONED vict_age > 90
Performing operation for this query: PRESENT area_name, crm_cd_desc THIS la_crime_dataset MENTIONED vict_age > 90
Final Result
| Index |   area_name   |                      crm_cd_desc                       |
| 0     | Hollenbeck    |    THEFT FROM MOTOR VEHICLE - PETTY ($950 & UNDER)     |
| 1     |   Olympic     |  THEFT FROM MOTOR VEHICLE - GRAND ($950.01 AND OVER)   |
| 2     | Devonshire    |    THEFT FROM MOTOR VEHICLE - PETTY ($950 & UNDER)     |
| 3     |  Foothill     |                BURGLARY FROM VEHICLE                   |
| 4     |  Van Nuys     |              BATTERY - SIMPLE ASSAULT                  |
| 5     |  Van Nuys     |       VANDALISM - MISDEAMEANOR ($399 OR UNDER)         |
| 6     | Devonshire    |                       BURGLARY                         |
| 7     |   Topanga     |          THEFT PLAIN - PETTY ($950 & UNDER)            |
| 8     |  Van Nuys     |                       BURGLARY                         |
| 9     |   Topanga     |        LETTERS LEWD  -  TELEPHONE CALLS LEWD           |
| 10    | West Valley   |                       BURGLARY                         |
| 11    |   Topanga     |                       BURGLARY                         |
| 12    |  Van Nuys     |                       BURGLARY                         |
| 13    |   Harbor      |                       BURGLARY                         |
| 14    | N Hollywood   |     ASSAULT WITH DEADLY WEAPON AGGRAVATED ASSAULT      |
| 15    |   Harbor      | VANDALISM - FELONY ($400 & OVER ALL CHURCH VANDALISMS) |
| 16    | 77th Street   |                  CONTEMPT OF COURT                     |
| 17    |   Olympic     |               BURGLARY FROM VEHICLE                    |
| 18    |  Hollywood    |                       ROBBERY                          |
| 19    |   West LA     |               BURGLARY FROM VEHICLE                    |
| 20    |   West LA     | THEFT-GRAND ($950.01 & OVER)EXCPTGUNSFOWLLIVESTKPROD   |
| 21    | Hollenbeck    |                    _TRESPASSING                        |
```

Order by

Objective: Sort CSV data chunks by a specified column, then combine the sorted chunks into a single CSV file. It also removes temporary files created during the sorting and merging processes.

Overview of Functionality:

- **Reading and Sorting Data Chunks:**
  - Reads data from a CSV file in chunks using **read_csv_in_chunks**.
  - Use **apply_order_by** to sort each chunk based on the specified column and sort order.
  - The sorted chunks are saved in a temporary folder as separate CSV file.
- **Sorted File Merging:**
  - The **merge_sorted_csv** function combines two sorted CSV files into a single sorted file.
  - It reads two CSV files line by line, comparing and merging them according to the column and sort order specified.
  - Merged files are combined sequentially until a final merged CSV is obtained.
- **Final Cleanup and Result Generation:**
  - After merging all sorted files, a final merged CSV file is obtained by **merge_and_cleanup_files** fucntion
  - Temporary and result directories are managed:
    - Temporary files from the sorting process are deleted.
    - Any existing result files in the result directory are removed.
    - The final merged CSV file is moved to the result directory.

```
Enter your SQL query (Type 'exit' to quit): PRESENT vict_sex THIS la_crime_dataset MENTIONED area_name = 'Harbor' SORT vict_age STRAIGHT
Performing operation for this query: PRESENT vict_sex THIS la_crime_dataset MENTIONED area_name = 'Harbor' SORT vict_age STRAIGHT
Final Result
| Index | vict_sex |
| 0     |    F     |
| 1     |    M     |
| 2     |    M     |
| 3     |    F     |
| 4     |    F     |
| 5     |    F     |
| 6     |    F     |
| 7     |    M     |
| 8     |    F     |
| 9     |    M     |
| 10    |    M     |
| 11    |    F     |
| 12    |    F     |
| 13    |    M     |
| 14    |    F     |
| 15    |    M     |
| 16    |    M     |
| 17    |    M     |
```

Grouping

Objective: The code aims to perform a 'group by' operation on a specified column from a CSV file. It identifies unique values in the specified column and creates separate chunk files.
Overview of Functionality:

- **Recognizing Distinctive Values:**
  - Reads data from a CSV file in chunks using read_csv_in_chunks.
  - If the operation is a 'groupby' operation:
  - Returns the specified column (specified_column) for grouping.
  - Iterates over data chunks to extract and store unique values from the specified column in unique_values.
- **Data Grouping:**
  - Once unique values are identified:
  - Iterates over the data chunks again for each unique value.
  - Determines whether a file named 'groupby_result.csv' exists.
  - Filters rows from data chunks based on the current unique value and appends the filtered rows to 'groupby_result.csv' or creates the file if it does not already exist.
  - Once all data has been grouped into 'groupby_result.csv,' move 'groupby_result.csv' to the'result' folder as'result.csv'.
  - Removes temporary files and folders created during grouping

```
Enter your SQL query (Type 'exit' to quit): PRESENT MEAN(vict_age) THIS la_crime_dataset CLUSTER area_name
Performing operation for this query: PRESENT MEAN(vict_age) THIS la_crime_dataset CLUSTER area_name
| Index | vict_age_custom_average |  area_name   |
| 0     |    38.58211143695015    | N Hollywood  |
| 1     |    38.81288343558282    | Devonshire   |
| 2     |    43.22107969151671    | West Valley  |
| 3     |    40.63736263736264    |   Pacific    |
| 4     |    40.86089238845145    |  Wilshire    |
| 5     |    39.86111111111112    |   West LA    |
| 6     |    37.33029197080292    |   Harbor     |
| 7     |    37.80335731414869    | 77th Street  |
| 8     |   40.398305084745765    |  Southwest   |
| 9     |    37.68421052631579    |   Olympic    |
| 10    |        38.28125         |   Topanga    |
| 11    |    39.34090909090909    |   Central    |
| 12    |    41.27097902097902    |  Hollywood   |
| 13    |    37.8609865470852     |   Mission    |
| 14    |    37.36926147704591    |   Newton     |
| 15    |    36.51639344262295    | Hollenbeck   |
| 16    |    41.30107526881721    |  Foothill    |
| 17    |    42.36590909090909    |  Van Nuys    |
| 18    |    42.19214876033058    |  Southeast   |
| 19    |    42.73493975903615    |  Northeast   |
| 20    |   39.976470588235294    |   Rampart    |
Enter your SQL query (Type 'exit' to quit):
```

Aggregation

Overview of Functionality:

- **custom_aggregation**
  - Defines a number of custom aggregation functions (**custom_sum, custom_min, custom_max, custom_average, custom_count**) that operate on DataFrame chunks to compute the aggregation results.
- **Aggregation Handler:**
  - Assumes that the dataset is already grouped by some column
  - **aggregate_handler** receives aggregate functions and, if specified, a group-by column.
  - It converts the provided aggregate functions into custom aggregation functions.
  - For each aggregation function specified:
  - Based on the function type and column, calls the appropriate custom aggregation function (**custom_aggregation**).
  - Appends the result to a list of aggregate columns (result).
  - **custom_aggregation** is the core function responsible for the aggregation process.
  - Iterates through the CSV chunks in the result folder.
  - For each chunk, calls the corresponding custom aggregation functions and aggregates the values based on groupings.
  - Creates a DataFrame containing aggregated values.
- **File Handling and Finalization:**
  - Handles the 'aggregate' operation by orchestrating the aggregation process.
  - Returns aggregate functions and, if specified, the group-by column.
  - Aggregation is performed using the **aggregate_handler,** which generates a final aggregated (df).
  - Adds the group-by column to the DataFrame if desired.
  - Removes any existing files from the result folder.
  - Saves the resulting DataFrame as'result.csv' in the result folder or handles single-row aggregation based on the presence of a group by column

```
Enter your SQL query (Type 'exit' to quit): PRESENT LEAST(vict_age) THIS la_crime_dataset CLUSTER area_name
Performing operation for this query: PRESENT LEAST(vict_age) THIS la_crime_dataset CLUSTER area_name
| Index | vict_age_custom_min |  area_name  |
| 0     |         4.0         | N Hollywood |
| 1     |        12.0         | Devonshire  |
| 2     |         6.0         | West Valley |
| 3     |         5.0         |   Pacific   |
| 4     |         2.0         |  Wilshire   |
| 5     |         7.0         |   West LA   |
| 6     |         8.0         |   Harbor    |
| 7     |         5.0         | 77th Street |
| 8     |         6.0         |  Southwest  |
| 9     |         5.0         |   Olympic   |
| 10    |         6.0         |   Topanga   |
| 11    |         4.0         |   Central   |
| 12    |         3.0         |  Hollywood  |
| 13    |         4.0         |   Mission   |
| 14    |         2.0         |   Newton    |
| 15    |         5.0         | Hollenbeck  |
| 16    |         8.0         |  Foothill   |
| 17    |         5.0         |  Van Nuys   |
| 18    |        14.0         |  Southeast  |
| 19    |         7.0         |  Northeast  |
| 20    |         2.0         |   Rampart   |
```

Update

Overview of Functionality:

- perform_update_query_on_chunk(parsed_query, data_chunk,buffer_folder_path)
    - Executes update operations on data chunks rather than the entire dataset at once.
    - Modifies specific rows within a chunk based on the conditions provided.
    - Saves the updated chunk as a temporary file in a buffer folder.
- The update operation reads and modifies rows in the CSV file using Python's CSV module. Based on user-defined conditional logic, it identifies rows to update and updates specific columns with new values.

Insert

Overview of Functionality:

- insert_query_on_chunk_perform(parsed_query, data_chunk, buffer_folder_path)
    - Operates on data chunks, inserting a new row into the dataset's final chunk.
    - Implementation: The insert operation uses the CSV module to append new rows to the end of the CSV file, ensuring that the existing dataset remains intact.

Delete

Overview of Functionality:

- query_delete_on_chunk(parsed_query, data_chunk, buffer_folder_path)
    - Operates on data chunks, removing rows that meet the specified conditions.
    - Removes rows from a chunk based on some criteria.

Tech Stack

1. **Python:** The core programming language driving the entire project.
2. **Ply (Python Lex-Yacc):** which are especially useful for interpreting and parsing Custom Query Language.
3. **Pandas:** Leveraged for certain data manipulation.
4. **NumPy:** Utilized for fundamental mathematical computations like avg, sum, etc
5. **CSV:** facilitated reading and writing CSV files.
6. **Shutil & OS:** managing files and directories, including operations like moving, copying, and deleting files. This could have been crucial for handling temporary files, creating folders, or managing project-specific directory structures.
7. **Regular Expressions (re):** useful for parsing text, extracting specific patterns, or performing search-and-replace operations based on defined patterns within strings.

Learning Outcomes

1. **Chunk-Based Processing**: Recognizing the value of chunk-based processing for large datasets, optimizing memory usage, and enabling efficient operations on large CSV files.
2. **Modular Code Design:** Discover the advantages of modular code design for improved readability, maintainability, and scalability, as well as promoting function reusability across different operations.
3. **SQL Query Handling:** Developing proficiency in parsing and executing SQL queries on CSV data, allowing users to perform a variety of data manipulation tasks.
4. **Error Handling and Validation**: Recognizing the significance of robust error handling and data validation mechanisms in maintaining data integrity and preventing potential problems.
5. **Performance Optimization:** Investigating ways to improve code performance, such as hashing, condition optimization, etc.

Challenges

1. **Memory Management:** Dealing with memory constraints while working with large datasets, which necessitates the use of chunk-based processing to prevent memory overflow.
2. **Complexity of Error Handling:** Addressing diverse error scenarios and implementing robust error handling mechanisms, particularly when dealing with diverse CSV structures or malformed queries.
5**. Scalability Issues:** Addressing scalability issues for various data sizes, with the goal of achieving consistent performance across small and large datasets without sacrificing efficiency.

Conclusion

The project successfully implements diverse functionalities for defining and manipulating large csv datasets efficiently.

Key learnings include implementing the concepts taught in class and getting a practical feel behind the theory which taught me the strategies for chunk-based operations, and enhancing code scalability for managing sizable datasets.

This report emphasizes the project's effectiveness in implementing functionalities for data processing, while also acknowledging challenges encountered and learnings gained throughout the development process.

Future Scope

Error Handling and Validation

1. Error Detection: Improve error handling mechanisms to detect and gracefully handle a variety of error scenarios, such as invalid query inputs, missing columns, and file corruption.

2. Data Integrity Checks: Use robust validation to ensure data integrity, preventing incorrect or malformed data from being inserted.

Improved Functionality

1.**More Functionality**: Expand SQL-like query support to include a broader range of SQL operations and functions,

allowing users to execute a broader range of queries.

User Experience

1. **Interactive Interfaces:** Create user-friendly interfaces for interacting with operations, allowing users to execute queries and view results in a seamless manner.

2. **Feedback and Logging:** Include mechanisms for tracking query executions, error logs, and performance metrics, as well as logging.

Integration and Compatibility

1. **File Formats:** Extend compatibility to support file formats other than CSV, such as Excel, JSON, and database connections.

Integration of Machine Learning