Naïve DB – a simple Database system

Final Report

Foundations of Data Management by Professor. Wensheng Wu

Harshit Kumar Jain
M.S Computer Science
https://www.linkedin.com/in/harshitkumarjain/
https://constharshit.github.io/

Table of Contents

Introduction

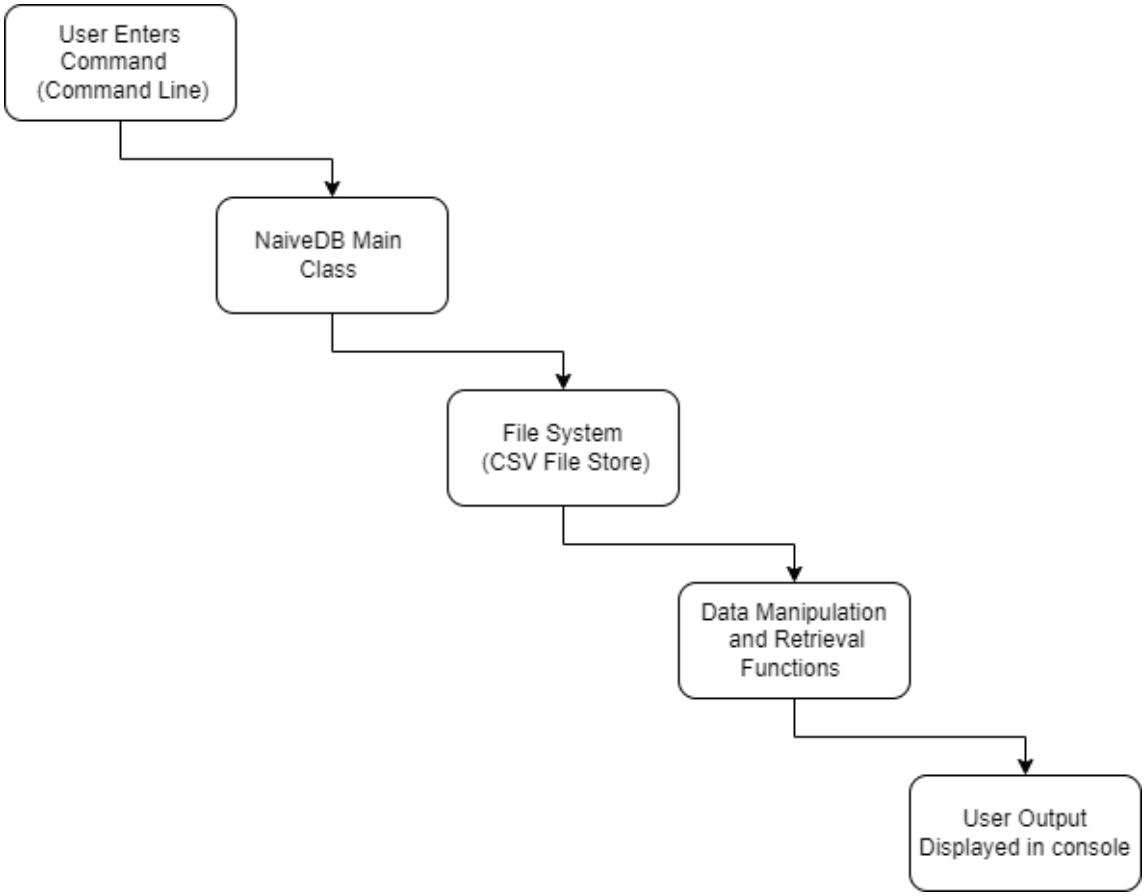
In this Project, the goal is to implement a database system that has its own query language, which must be different from the existing query languages. The main challenge here is that we can only simultaneously load part of the dataset in the main memory. This calls for handling the dataset in chunks like real-world database systems like MySQL handle large datasets.

Planned Implementation

In our implementation, we have assumed that our dataset can only load 500 rows in the main memory at any given time. The minor part was implementing the query language, which depends on how complicated and sophisticated the system you want to build. Our dataset has 5000 rows, and our memory is 500 rows. We have implemented the NaiveDB in such a way that it demonstrates the CRUD operations, projection, filtering, join, grouping, aggregation, and ordering.

NaiveDB performs operations on existing datasets or tables stored as CSV files. It also needs a column named id, which is the primary key in any table. The main challenges were implementing the join, grouping, and ordering operations, as we could only load part of the dataset into the main memory.

Architecture Design



Description of Design

After many hours of thinking and trying out different things, we decided to stick to the essential interaction with the user through a console where the program asks for the query from the user through the console, parses, performs the required function and returns the value to the console to inform the user of the output.

The flow of the program is as follows:

1. The user enters the query.
2. Our program parses it and infers the command; we have all the data to operate.
3. The program is designed such that with the user entered, data goes to the respective operations function, and the query is thus executed.
4. The user is notified of the output through the console.

We were required to use a real-world dataset from sources like Kaggle and Google. We have used the Metal Bands dataset from Kaggle.
Here is the link :
https://www.kaggle.com/datasets/mrpantherson/metal-by-nation
It has 5000 rows and the columns id, name, band_name, formed, origin, split, and style. The dataset had many null values cleaned using a Python script.

NaiveDB uses a simple language and nothing fancy to avoid confusing the user. The query is separated by the '|' for easy parsing. The first part of the query is always a command. An example of how the queries look like is this:

Enter your query in the following format:

1. CREATE TABLE: newTable|table_name|column1,column2,...

2. INSERT: addToTable|table_name|col1_val,col2_val,...

3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all

4. ORDER BY: sort|table_name|col_name

5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val

6. DELETE: remove|table_name|col_name|col_value

7. GROUP BY: formGroups|table_name|col_name

8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)

9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2

10. AGGREGATE: aggregate|table_name|col_name|operation

11. Type bye to exit

NaiveDB >

Functionalities and Tech Stack

Python:
Core programming language for implementing the database operations.

CSV:
File format for storing table data.
Git:
Version control for tracking changes in the codebase.

Visual Studio Code (or any Python IDE):
Development environment for writing and testing the code.

FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)

1. Check if the CSV file corresponding to the input table name exists. If not, print an error message and exit.

2. Create a new CSV file named "output.csv" for storing the filtered data.

3. Define a filtering condition based on the specified condition type ("equalTo," "smallerThan," or "biggerThan"). If the type is invalid, print an error message and exit.

4. Read the original CSV file in chunks of 500 rows.

5. For each chunk, open the new CSV file in append mode and write the header if it's a new file. Iterate through the rows in the chunk, applying the filtering condition, and write the matching rows to the new file.

6. Open the new CSV file and read its rows as dictionaries using `csv.DictReader`. Print each row to display the filtered data.

7. Remove the temporary output CSV file ("output.csv").

UPDATE:
set|table_name|col_name|cond_val|update_col|new_val

1. Check if the CSV file corresponding to the input table name exists. If not, print an error message and exit.

2. Define the default ID column as "id."

3. If the condition column is the ID column, check for duplicates of the update value in the ID column. If a duplicate is found, print an error message and exit.

4. Create a temporary output file named "temp_updated.csv" for storing the updated data.

5. Initialize variables to track the number of rows processed and updated.

6. Open the original CSV file for reading and the temporary output file for writing.

7. Read the CSV file row by row using a DictReader and write the header to the temporary output file.

8. Process each row, checking if the condition column's value matches the specified condition value. If a match is found, update the specified update column's value.

9. Keep track of the processed and updated rows.

10. Write the processed rows to the temporary output file in chunks of the specified size.

11. Remove the original CSV file.

12. Rename the temporary output file to the original CSV file, effectively replacing it.

13. Print a summary of the update operation, indicating the number of rows updated out of the total number of rows processed.

JOIN:
getCommon|table1_name|table2_name|col_name_table1|col_name_table2

1. Confirm existence of both tables. Exit if one or more does not exist.

2. Open the first CSV file for reading.

3. Read data from the first CSV file in chunks, with chunk size specified by `chunk_size`.

4. For each chunk of rows from the first table, invoke the `process_chunk` method to manage the join with the second table based on the specified join column.

Process Chunk Method:

1. Accepts a chunk of rows from the first table, the name of the second table, and the join column in the second table.

2. For each row in the chunk, execute the join with the second table based on the indicated join column.

3. The specifics of the join operation are defined within the `process_chunk` method. The method's implementation would involve correlating rows between the first and second tables based on the join column and either displaying or storing the combined rows.

AGGREGATE:
aggregate|table_name|col_name|operation

1. Verify the existence of the specified CSV file. If it doesn't exist, print an error message and exit.

2. Convert the specified operation to lowercase for case-insensitive comparison. If the operation is not one of ('average', 'sum', 'minimum', 'maximum'), print an error message and exit.

3. Initialize variables for the aggregation result and row count.

4. Open the CSV file for reading.

5. Iterate through each row in the CSV file. For each row, check if the specified column exists.

6. If the column exists, convert the column's value to an integer and perform the specified aggregation operation ('average', 'sum', 'minimum', 'maximum').

7. Track the number of rows processed.

8. After processing all rows, if there is a valid result, calculate the final value for 'average' and print the result with the specified operation and column name.

9. If no valid result is obtained, print a message indicating no data found in the specified column.

10. Handle exceptions for file not found and value conversion errors.

Note: The function aggregates data from a specified column in a CSV file based on a specified operation and prints the result. It uses chunk processing with a default chunk size of 500 rows. The aggregation operation can be 'average', 'sum', 'minimum', or 'maximum'.

GROUP BY:
formGroups|table_name|col_name

1. Confirm the existence of the specified CSV table. If it doesn't exist, print an error message and exit.

2. Create a temporary folder named "temp" to store intermediate grouped data files.

3. Use a defaultdict to organize grouped chunks of rows based on the specified group column.

4. Open the CSV file for reading and process the rows in chunks.

5. Normalize the group column values to ensure consistent grouping.

6. Append rows to the current chunk until the chunk size is reached.

7. For each chunk, group the rows by the specified group column and store them in the defaultdict.

8. Sort and write each group of rows to temporary CSV files within the "temp" folder.

9. Create a list of sorted temporary file names.

10. Open the specified output file for writing and write the grouped data.

11. Print the grouped data.

12. Print a message indicating that the grouped data has been saved to the specified output file.

13. Handle exceptions and print an error message if any occur during the process.

DELETE:
remove|table_name|col_name|col_value

1. Confirm the existence of the specified CSV table. If it doesn't exist, print an error message and exit.

2. Define a new CSV file named "output.csv" for storing data without rows matching the specified condition.

3. Iterate through the CSV file in chunks of 500 rows using the `read_csv_in_chunks` method.

4. For each chunk, open the new CSV file in append mode, write the header if it's a new file, and iterate through the rows. Write rows to the new file only if they do not match the specified condition.

5. Clear the contents of the original CSV file.

6. Iterate through the new CSV file in chunks again.

7. For each chunk, open the original CSV file in append mode, write the header if it's a new file, and write the rows from the chunk to the original CSV file.

8. Remove the temporary output CSV file.

Note: The function deletes rows from a CSV table based on a specified condition by creating a new CSV file without the matching rows and then replacing the original dataset with the new data. The condition is specified by a function that takes a row as input and returns a boolean value indicating whether to delete the row.

ORDER BY:
sort|table_name|col_name

1. Confirm the existence of the specified CSV table. If it doesn't exist, print an error message and exit.

2. Create a temporary folder named "temp_order" to store temporary files during the sorting process.

3. Define a helper function `int_or_original` to attempt conversion to an integer and fallback to the original value if conversion fails.

4. Phase 1: Divide and sort chunks
   - Initialize variables for chunk count, fieldnames, and temporary files.
   - Open the CSV file for reading and iterate through rows.
   - Sort and write chunks to temporary files based on the specified order column.

5. Phase 2: Merge sorted chunks using external merge sort
   - Create iterators for each temporary file.
   - Merge sorted chunks into a list using `heapq.merge` based on the order column.

6. Clear the output file if it exists and write the ordered data to the specified output file.

7. Print the ordered data.

8. Print a message indicating that the data has been ordered by the specified column and saved to the output file.

9. Handle exceptions and print an error message if any occur during the sorting process.

SELECT:
showColumns|table_name|col1,col2,... or showColumns|table_name|all

1. Confirm the existence of the specified CSV table. If it doesn't exist, print an error message and exit.

2. Open the CSV file for reading.

3. If specified columns are provided and they are not a subset of the table's columns, print an error message and exit.

4. If specified columns exist and are valid, print the column names separated by tabs.

5. Iterate through the rows in the CSV file.

6. If specified columns are provided, extract values for the specified columns and format them as a tab-separated string.

7. If no specified columns are provided, format all values in the row as a tab-separated string.

8. Print the formatted rows in chunks based on the specified chunk size.

9. Handle cases where the number of rows is not a multiple of the chunk size to ensure all rows are printed.

Note: The function selects and prints rows from a CSV table with optional column filtering. It prints the selected rows in chunks based on the specified chunk size. If no columns are specified, all columns are selected. If specified columns are provided, the function checks for their validity before printing.

INSERT:
addToTable|table_name|col1_val,col2_val,...

1. Confirm the existence of the specified CSV table. If it doesn't exist, print an error message and exit.

2. Define the ID column as "id" and extract the target ID from the provided values.

3. Check for duplicate IDs using the `check_for_duplicate_id` method. If a duplicate ID is found, print a message and exit.

4. Open the CSV file for appending and write the provided values as a new record.

5. Print a message indicating that the record has been inserted into the specified table.

Method to check for duplicate id:

1. Check if the CSV file for the specified table exists. If it doesn't exist, return False (no duplicate ID).

2. Open the CSV file for reading and iterate through rows.

3. Check if the specified ID column exists in the row and if its value matches the target ID.

4. If a match is found, return True (duplicate ID).

5. If no match is found after iterating through all rows, return False (no duplicate ID).

Note: The `insert` function inserts a new record into a specified table, checking for duplicate IDs before insertion. The `check_for_duplicate_id` method is a helper method used to check for duplicate IDs in the specified table.

CREATE TABLE:
newTable|table_name|column1,column2,...

1. Define the CSV file name based on the provided table name.

2. Check if the CSV file already exists. If it does, print a message and exit.

3. Open the CSV file for writing and create a `DictWriter` object with the specified column names.

4. Write the header to the CSV file.

5. Print a message indicating that the table has been created with the specified columns.

Note: The function creates a new CSV table with the given name and columns if it doesn't already exist. It uses the `csv.DictWriter` class to write the header with the specified column names to the CSV file.

Implementation Screenshots

Here are some snips of the program:

Join operation:

Joining two tables s1 and s2 based on id column of s1 and id column of s2.

NaiveDB > getCommon|s1|s2|id|id

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > getCommon|s1|s2|id|id
{'table1_id': '2', 'table1_band_name': 'Metallica', 'table1_fans': '3712', 'table1_formed': '1981', 'table1_origin': 'USA', 'table1_split': '2017', 'table1_style': 'Hip Hop', 'tab
le2_id': '2', 'table2_band_name': 'Metallica', 'table2_fans': '3712', 'table2_formed': '1981', 'table2_origin': 'USA', 'table2_split': '2017', 'table2_style': 'Heavy,Bay area thra
sh'}
{'table1_id': '3', 'table1_band_name': 'Megadeth', 'table1_fans': '3105', 'table1_formed': '1983', 'table1_origin': 'USA', 'table1_split': '1983', 'table1_style': 'Thrash,Heavy,Ha
rd rock', 'table2_id': '3', 'table2_band_name': 'Megadeth', 'table2_fans': '3105', 'table2_formed': '1983', 'table2_origin': 'USA', 'table2_split': '1983', 'table2_style': 'Thrash
,Heavy,Hard rock'}
{'table1_id': '4', 'table1_band_name': 'k', 'table1_fans': '67', 'table1_formed': '1984', 'table1_origin': 'INdia', 'table1_split': '9089', 'table1_style': 'Classical', 'table2_id
': '4', 'table2_band_name': 'Amon Amarth', 'table2_fans': '3054', 'table2_formed': '1988', 'table2_origin': 'Sweden', 'table2_split': '2017', 'table2_style': 'Melodic death'}
```

Aggregate operation:

NaiveDB > aggregate|s2|fans|maximum

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > aggregate|s2|fans|maximum
Maximum of fans: 4195
```

Filter operation:

NaiveDB > filter|metal_bands_2017|fans|4000|biggerThan

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > filter|metal_bands_2017|fans|4000|biggerThan
{'id': '0', 'band_name': 'Iron Maiden', 'fans': '4195', 'formed': '1975', 'origin': 'United Kingdom', 'split': '2017', 'style': 'New wave of british heavy,Heavy'}
{'id': '1', 'band_name': 'Opeth', 'fans': '4147', 'formed': '1990', 'origin': 'Sweden', 'split': '1990', 'style': 'Extreme progressive,Progressive rock,Progressive'}
{'id': '50', 'band_name': 'Iron Maiden', 'fans': '4195', 'formed': '1975', 'origin': 'United Kingdom', 'split': '2017', 'style': 'New wave of british heavy,Heavy'}
{'id': '51', 'band_name': 'Opeth', 'fans': '4147', 'formed': '1990', 'origin': 'Sweden', 'split': '1990', 'style': 'Extreme progressive,Progressive rock,Progressive'}
{'id': '4999', 'band_name': 'Axis Of Despair', 'fans': '323232', 'formed': '2014', 'origin': 'Sweden', 'split': '2014', 'style': 'Grindcore'}
```

Grouping operation:

NaiveDB > formGroups|s1|origin

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > formGroups|s1|origin
{'id': '0', 'band_name': 'Iron Maiden', 'fans': '4195', 'formed': '1975', 'origin': 'United Kingdom', 'split': '2017', 'style': 'New wave of british heavy,Heavy'}
{'id': '8', 'band_name': 'Black Sabbath', 'fans': '2307', 'formed': '1968', 'origin': 'United Kingdom', 'split': '2017', 'style': 'Doom,Heavy,Hard rock'}
{'id': '1', 'band_name': 'Opeth', 'fans': '4147', 'formed': '1990', 'origin': 'Sweden', 'split': '1990', 'style': 'Extreme progressive,Progressive rock,Progressive'}
{'id': '4', 'band_name': 'Amon Amarth', 'fans': '3054', 'formed': '1988', 'origin': 'Sweden', 'split': '2017', 'style': 'Melodic death'}
{'id': '2', 'band_name': 'Metallica', 'fans': '3712', 'formed': '1981', 'origin': 'USA', 'split': '2017', 'style': 'Heavy,Bay area thrash'}
{'id': '3', 'band_name': 'Megadeth', 'fans': '3105', 'formed': '1983', 'origin': 'USA', 'split': '1983', 'style': 'Thrash,Heavy,Hard rock'}
{'id': '5', 'band_name': 'Slayer', 'fans': '2955', 'formed': '1981', 'origin': 'USA', 'split': '1981', 'style': 'Thrash'}
{'id': '6', 'band_name': 'Death', 'fans': '2690', 'formed': '1983', 'origin': 'USA', 'split': '2001', 'style': 'Progressive death,Death,Progressive thrash'}
{'id': '7', 'band_name': 'Dream Theater', 'fans': '2329', 'formed': '1985', 'origin': 'USA', 'split': '1985', 'style': 'Progressive'}
Grouped data saved to output1.csv.
```

Delete operation:

NaiveDB > remove|s1|id|2

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > remove|s1|id|2
Delete complete.
```

Update operation:

NaiveDB > set|s1|id|7|origin|India

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > set|s1|id|7|origin|India
Updated 1 out of 8 rows in 's1.csv'.
Update complete.
```

Ordering operation:

NaiveDB > sort|s1|fans

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > sort|s1|fans
{'id': '8', 'band_name': 'Black Sabbath', 'fans': '2307', 'formed': '1968', 'origin': 'United Kingdom', 'split': '2017', 'style': 'Doom,Heavy,Hard rock'}
{'id': '7', 'band_name': 'Dream Theater', 'fans': '2329', 'formed': '1985', 'origin': 'India', 'split': '1985', 'style': 'Progressive'}
{'id': '6', 'band_name': 'Death', 'fans': '2690', 'formed': '1983', 'origin': 'USA', 'split': '2001', 'style': 'Progressive death,Death,Progressive thrash'}
{'id': '5', 'band_name': 'Slayer', 'fans': '2955', 'formed': '1981', 'origin': 'USA', 'split': '1981', 'style': 'Thrash'}
{'id': '4', 'band_name': 'Amon Amarth', 'fans': '3054', 'formed': '1988', 'origin': 'Sweden', 'split': '2017', 'style': 'Melodic death'}
{'id': '3', 'band_name': 'Megadeth', 'fans': '3105', 'formed': '1983', 'origin': 'USA', 'split': '1983', 'style': 'Thrash,Heavy,Hard rock'}
{'id': '1', 'band_name': 'Opeth', 'fans': '4147', 'formed': '1990', 'origin': 'Sweden', 'split': '1990', 'style': 'Extreme progressive,Progressive rock,Progressive'}
{'id': '0', 'band_name': 'Iron Maiden', 'fans': '4195', 'formed': '1975', 'origin': 'United Kingdom', 'split': '2017', 'style': 'New wave of british heavy,Heavy'}
Data ordered by 'fans' and saved to ordered.csv.
```

Projection operation:

NaiveDB > showColumns|s1|id,band_name,origin,fans

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > showColumns|s1|id,band_name,origin,fans
id      band_name       origin  fans
0       Iron Maiden     United Kingdom  4195
1       Opeth   Sweden  4147
3       Megadeth        USA     3105
4       Amon Amarth     Sweden  3054
5       Slayer  USA     2955
6       Death   USA     2690
7       Dream Theater   India   2329
8       Black Sabbath   United Kingdom  2307
```

Insert operation:

NaiveDB > addToTable|s1|20,USC,777,2022,USA,2024,Classical

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > addToTable|s1|20,USC,777,2022,USA,2024,Classical
Record inserted into table 's1'.
```

Create operation:

NaiveDB > newTable|s3|id,name,fans

```
Enter your query in the following format:
1. CREATE TABLE: newTable|table_name|column1,column2,...
2. INSERT: addToTable|table_name|col1_val,col2_val,...
3. SELECT: showColumns|table_name|col1,col2,... or showColumns|table_name|all
4. ORDER BY: sort|table_name|col_name
5. UPDATE: set|table_name|col_name|cond_val|update_col|new_val
6. DELETE: remove|table_name|col_name|col_value
7. GROUP BY: formGroups|table_name|col_name
8. FILTER: filter|table_name|col_name|col_value|condition(smallerThan/biggerThan/equalTo)
9. JOIN: getCommon|table1_name|table2_name|col_name_table1|col_name_table2
10. AGGREGATE: aggregate|table_name|col_name|operation
11. Type bye to exit
NaiveDB > newTable|s3|id,name,fans
Table 's3' created with columns: ['id', 'name', 'fans']
```

Learning Outcomes and Challenges Faced

There were numerous challenges throughout the journey of building the NaiveDB. One challenge was while implementing the sorting functionality wherein I was able to divide the data into chunks and sort the chunks successfully but I needed to know a way to merge the chunks, and they were outputting sorted chunks, but the whole output needed to be relatively sorted. This was handled using the external sorting algorithm.

One other major challenge was for join, wherein I needed to learn how to load and compare all the rows of Table 1 with all the rows of Table 2. This was solved using the Nested loop join explained in the class.

For inserting data into tables, I checked if the id already exists, and if yes, we prevent the insertion. Then we append the row to the csv file. So here while checking for duplicate id in the table, I did it row by row. This can be done more efficiently by using indexing, and I plan to do this in the extension of this project after the semester. The main task was memory management and this was handled.

For select functionality, I just handled it using chunks, handling 500 rows at once and moving on the next 500, as this involved only projection and not manipulating the data.

For ordering, this was difficult as I had to maintain the final output in an ordered state. I used heapq for merging and external sorting for creating the sorted chunks.

For grouping, I used something similar to ordering as I created temp files for each group, kept adding the respective rows to those files, and then wrote the output from those files to an output file from where we printed the rows.

For update and deletion, I just used the condition given by the user, excluded those rows for deletion, and wrote the other rows to a new file. For deletion, based on the condition, just updated the respective rows, wrote the output to a new file, and transferred the data from the new file to the original dataset.

For Filter, the same approach takes 500 rows, releases them, and moves to the next chunk.

For join, I used the nested loop join approach by taking a chunk of 500 rows, going over them, and if they match, then building a new row and printing them to the console.

Conclusion

In conclusion, NaiveDB effectively fulfills its role as an educational tool, providing a foundational understanding of database management. Its simplicity, coupled with hands-on functionalities, makes it accessible for learners, while potential improvements could elevate its practical utility in educational contexts. NaiveDB serves as a starting point for individuals embarking on their journey to comprehend the essentials of database systems.

Future scope

- Be able to take multiple lines as a single query and parse it to perform the functions.
- Support multiple types of joins.
- Be able to join more than two tables.
- Be able to select the column name the user likes for primary key.
- Design a frontend web application and take the query from there and process it and send back the result to the user.
- Beautify the output shown to the user.
- Improvise the aggregation, sort by adding additional functionality.