Final Report

Group Members: Xuesong Shen (xuesongs@usc.edu), Haoze Zhu (haozezhu@usc.edu)
Class Session: Tuesday Afternoon

Introduction

In our project, we implemented two database systems—one tailored for SQL databases and the other for NoSQL databases. These two database systems share one command interface and the same query language structure. Our team designed our own query language capable of supporting various operations such as Projection, Filtering, Join, Grouping, Aggregation, Ordering, Inserting, Deleting, and Updating. Moreover, we applied methods like Nested Loop Join and External Merge Sort to comply with the memory constraints. We also encountered several obstacles, including devising methods to read data in chunks, implementing functions for join and sort operations, and adapting queries for both CSV and JSON file formats. During the process, we gained a deeper insight into how database systems operate and how the functionalities of database systems are realized.

Here is the link to our code in Google Drive.

Planned Implementation

The basic design of the system: We plan to use Python to design and implement our systems. For Storage, we plan to use CSV files for the relational model and JSON files for NoSQL data.

Data model: For a relational model, data are represented in tables with rows and columns. Each table is stored in a CSV file. For a NoSQL model, data is represented as Collections and Documents. Each Collection is stored in a JSON file.

Ideas on how to implement them:

First, with different real-world data sets, we will create a relational database and a non-relational database. Then, we will create tables with CSV files and collections with JSON files.

Specifically, to avoid reading the entire dataset into the main memory, we will use loops and indexes to read the data in smaller chunks, execute our query to each chunk, and combine our query to a new table or a new collection.

We will implement commands for projection, filtering, joining, grouping, aggregation, and ordering, as well as inserting, deleting, and updating the data in our query language through our interactive command line interface. For example, we will store a set of reserved words such as “create”, “find”, “table”, etc. If a reserved word is detected, the systems will read the data,

marked by marks such as single quotes or parentheses. After that, the Python program will do the operation on the corresponding data.

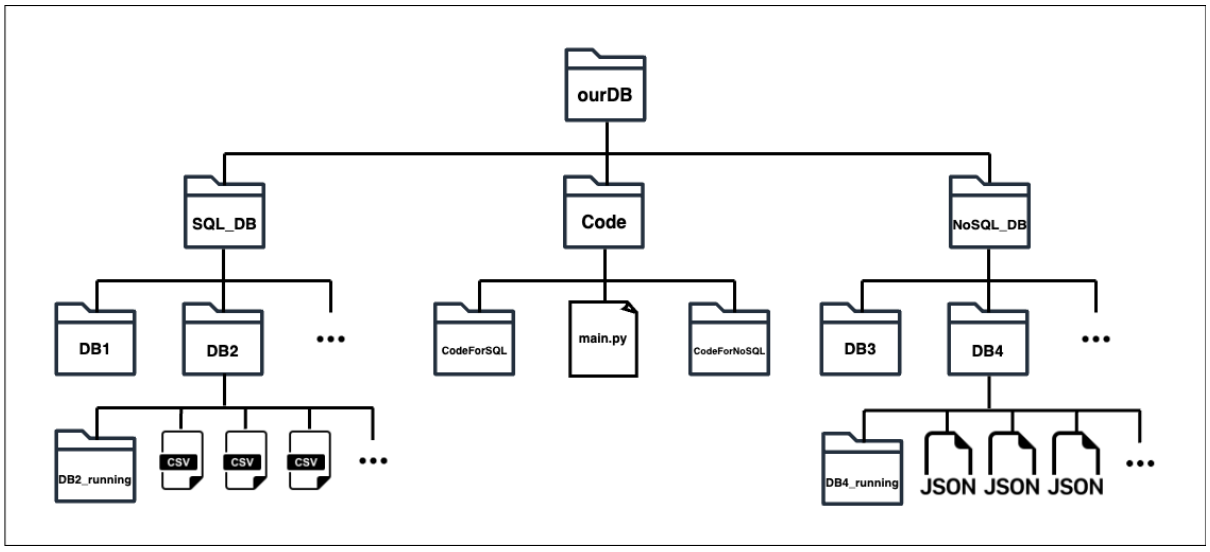
Architecture Design

System Structure

The root of our project is the *ourDB* folder. Under the root, there are three folders, *Code* for all code of the project, *SQL_DB* for SQL databases, and *NoSQL_DB* for NoSQL databases.

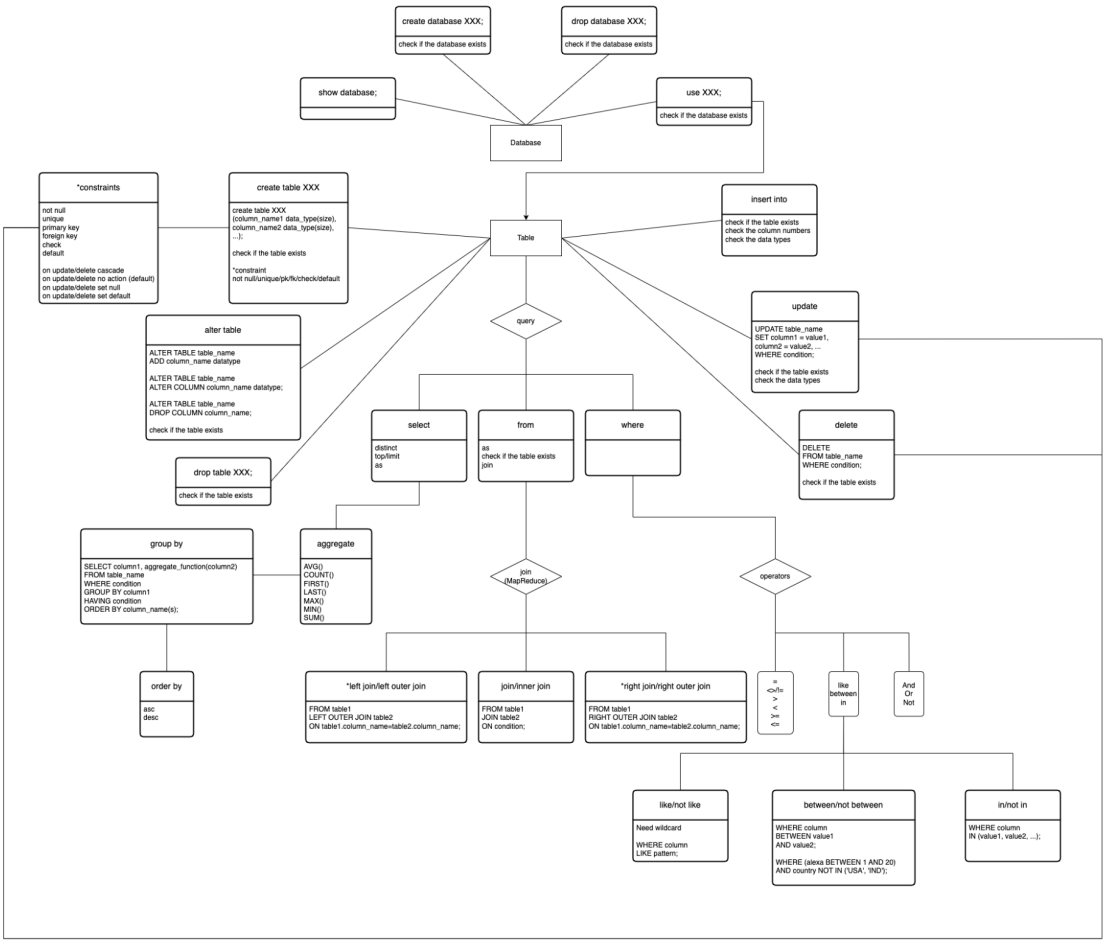
We use *main.py* to initialize our command interface, followed by choosing to use either SQL or NoSQL databases. Then we can select a specific database to operate the CRUD commands and queries.

Since we assume the main memory is much smaller than the files of the database, we use *_running* folders as disks to store the runtime files. Each database will have its own *_running* folder. Queries, like adding and updating data in a file, will create a temporary file in the *_running* folder, replacing the original file with the temporary file and deleting the extra file. The result of join, sort, and group operations will be stored in the *_running* folder, which could be used further in queries such as projection and filtering on the joined result. The content of the *_running* folder will be automatically removed while exiting the system to save system storage.



Functions:

We summarized the functions of MySQL as an example and implemented them in our database system.



Implementation

Tech Stack

1. **Programming Language: Python**

We chose Python for its ease of use and familiarity among team members. Its versatility and extensive library support made it ideal for our project's requirements.

2. **Libraries:**

- importlib, os, shutil: These libraries were crucial for handling system paths and managing file and folder operations like creation, replacement, and deletion.
- itertools.islice: This module was used for efficiently selecting specific lines in CSV files, aiding in data manipulation and access.
- re (Regular Expressions): The re library is used in text matching within our NoSQL database system, enabling us to match queries with its functions.
- json: only used for converting JSON objects into Python dictionaries, this library facilitated the handling and processing of JSON data within our system.
- csv: only used for reading one line of the CSV file into our system.

3. Development Tools and IDEs

Visual Studio Code: Chosen for its robust features, extensibility, and ease of use, Visual Studio Code served as our primary Integrated Development Environment (IDE).

Git: We utilized Git for version control, which was instrumental in managing our codebase, tracking changes, and collaborating effectively as a team.

4. Datasets

SQL

NBA games data (from Kaggle)

- games.csv: all games from the 2004 season to the last update with the date, teams, and some details like the number of points, etc.
- games_details.csv: details of games dataset, all statistics of players for a given game
- players.csv: players details (name)
- ranking.csv: ranking of NBA given a day (split into west and east on conference column)
- teams.csv: all teams of NBA

File Name	File Size
teams.csv	4 KB
players.csv	266 KB
ranking.csv	15.5 MB
games.csv	4.1 MB
games_details.csv	93.1 MB

Note: the main memory for the SQL database system is 4 MB.

NoSQL

Indonesia's Top E-Commerce Tweets (from Kaggle)
This dataset contains the tweets from the first tweet until April 2020 of top e-commerce unicorns in Indonesia namely Shopee, Tokopedia, Bukalapak, Lazada, and Blibli.

File Name	File Size
bliblidotcom_stats.json	31.4 MB
bliblidotcom_tweet.json	20.9 MB
bukalapak_stats.json	27.9 MB
bukalapak_tweet.json	17.7 MB
tokopedia_stats.json	17.5 MB

Note: the main memory for the NoSQL database system is 4 MB.

Functionalities

Our query language is designed to have a conversational style, almost chat-like format, and utilize plain English sentences or phrases.

General Format: Action + Details/Conditions + Contextual Clues

The queries are phrased more like natural language requests or questions. Contextual clues are used to infer details not explicitly stated. Examples of Actions: Show, Connect, Summarize, List, Add, Remove, Change. These actions correspond to traditional operations like select, join, aggregate, order, insert, delete, and update. When a query is made in our command interface, we recognize/match the first word/phrase of the query (case insensitive) to their corresponding functions. For example, when ‘show’ is detected at the beginning of the query, we will send the query string to projection and filtering.

The details of each query function format are the following:

Creating a Database

- Format: Create a new database named [database name];
- Example: Create a new database named NBA;

Using a Specific Database

- Format: Switch to database [database name];
- Example: Switch to database NBA;

Deleting a Database

- Format: Delete database [database name];
- Example: Delete database NBA;

Showing Databases

- Format: Databases;
- Example: Databases;

Creating a Table/Collection

- Format: Set up a new table named [table name] with columns [column names];
- Format: Set up a new collection named [collection name] with [json object];
- Example: Set up a new table named teams with columns team_id, city, owner;

Dropping a Table

- Format: Drop table/collection [table/collection name];
- Example: Drop table teams; drop collection teams;

Showing Tables/Collection

- Format: Tables;
- Format: collections;
- Example: Tables;

Retrieving Data (Projection & Filtering)

- Format: Show [column/field(s)]/[all] of [table/collection] where [condition(s)] [line m-n];
- Example: Show team_id, city of teams where city = LA line 2-10;

Connecting Tables (Nested Loop Join)

- Format: Connect [table1/collection1] with [table2/collection2] based on [common feature];
- Example: Connect teams with players based on team_id = team_id;

Grouping and Aggregation

- Format: Summarize [column/field] on [column/field]/[all] from [table/collection] using [aggregation];
- Example: Summarize yearfounded on teams_id from teams using min;
- Aggregation: avg/sum (numeric only); count/min/max

Sorting Data (External Merge Sort)

- Format: Sort [table/collection] by [column/field] in [asc/desc] order;
- Example: Sort teams by yearfounded in asc order;

Inserting Data

- Format: Add [row] to [table/collection];
- Example: Add John Doe, Guard to players;

Updating Data

- Format: Change [column/field] to [new value] for [table] with [condition];
- Example: Change position to forward for players with name = John Doe;

Deleting Data

- Format: Remove row with [condition] from [table];
- Example: Remove row with name = John Doe from players;

Implementation Screenshots

System operation

Change the current path to *"ourDB/Code"*, then run *"python3 main.py"*.

```
Welcome to ourDB. Please choose 'SQL' or 'NoSQL'. Type 'exit' to quit.
ourDB >
```

To enter the SQL database system, run the command *"sql"*;

```
Welcome to ourDB. Please choose 'SQL' or 'NoSQL'. Type 'exit' to quit.
ourDB > sql
SQL Handler active. Type 'exit' to return to the main menu.
SQL_DB >
```

To enter the NoSql database system, run the command *"nosql"*;

```
Welcome to ourDB. Please choose 'SQL' or 'NoSQL'. Type 'exit' to quit.
ourDB > nosql
nosql
NoSQL Handler active. Type 'exit' to return to the main menu.
NoSQL_DB >
```

Run the command *"databases;"* to see all databases in the database system.

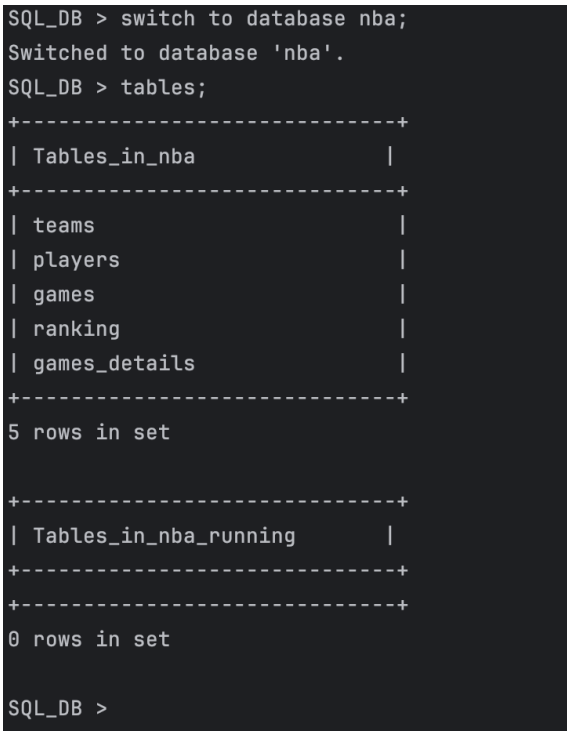
```
SQL_DB > Databases;
+--------------------+
| Database           |
+--------------------+
| cba                |
| nba                |
+--------------------+
2 rows in set

SQL_DB >
```

```
NoSQL_DB > databases;
+--------------------+
| Database           |
+--------------------+
| eco                |
| DatabaseB          |
+--------------------+
2 Database in set

NoSQL_DB >
```

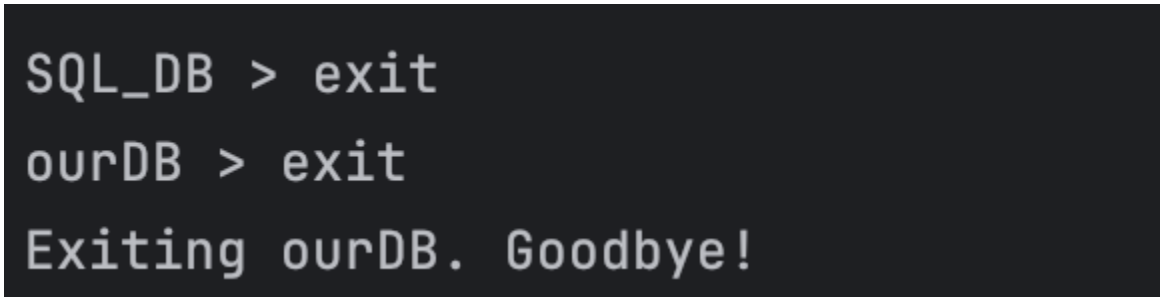
To use the database nba in the SQL databases, run the command *"switch to database nba;"*.
To show the tables in the database, run the command *"tables;"*.
To use the database eco in the NoSQL databases, run the command *"switch to database eco;"*.
To show the collections in the database, run the command "collections*;"*.



After switching to a specific database, one can use other query operations or CRUD commands on that database.

To exit the SQL database system, run the command *"exit"*.
All the runtime files in the running folders will be deleted at the same time.

To exit *ourDB*, run the command *"exit"*.



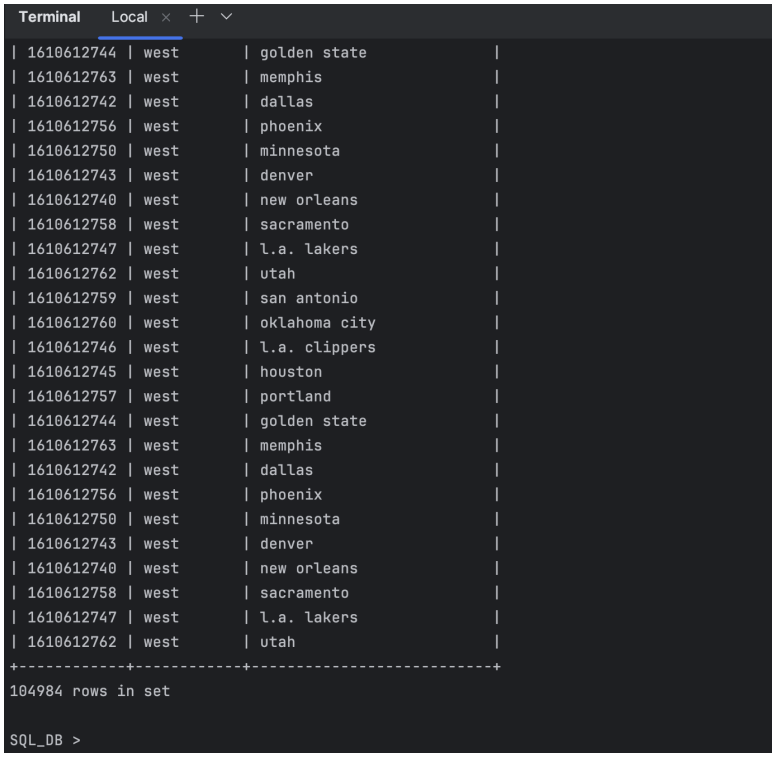
Query operations

Given the constraints on the length of this report, we are able to present only a select few screenshots that exemplify key aspects of our implementation. These screenshots specifically

illustrate the functionalities of *join, sort, projection* and *filtering,* as well as *grouping and aggregation* in our system.

SQL System: Retrieving Data (Projection & Filtering)

To show the table ranking with projection and filtering, run the command *"show team_id, conference, team of ranking where conference = west;"*.



To show the table with "limit" and "offset", run the command *"show team_id, conference, team of ranking where conference = west line 5-10;"*.

```
SQL_DB > show team_id, conference, team of ranking where conference = west line 5-10;
+------------+------------+--------------------------+
| team_id    | conference | team                     |
+------------+------------+--------------------------+
| 1610612746 | west       | la clippers              |
| 1610612758 | west       | sacramento               |
| 1610612762 | west       | utah                     |
| 1610612757 | west       | portland                 |
| 1610612742 | west       | dallas                   |
| 1610612750 | west       | minnesota                |
+------------+------------+--------------------------+
6 rows in set

SQL_DB >
```

SQL System: Connecting Tables (Nested Loop Join)

To connect two tables based on a common feature, run the command *"Connect teams with ranking based on team_id = team_id;"*.

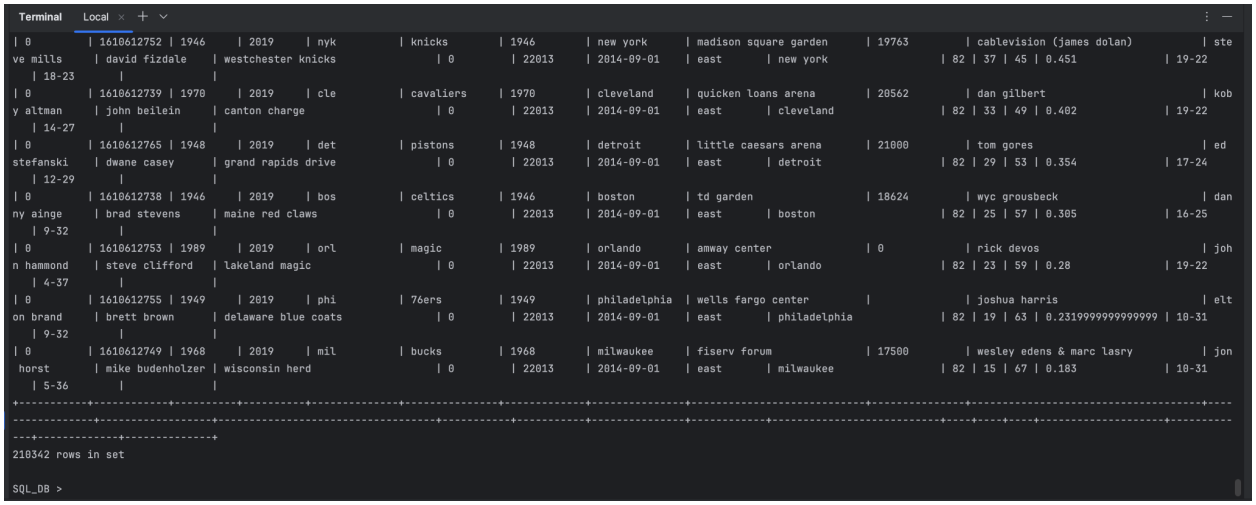
```
SQL_DB > Connect teams with ranking based on team_id = team_id;
Connected tables saved to ../SQL_DB/nba/nba_running/teams_connect_ranking_running.csv

SQL_DB > tables;
+------------------------------+
| Tables_in_nba                |
+------------------------------+
| teams                        |
| players                      |
| games                        |
| ranking                      |
| games_details                |
+------------------------------+
5 rows in set

+------------------------------+
| Tables_in_nba_running        |
+------------------------------+
| show_ranking_running         |
| teams_connect_ranking_running |
+------------------------------+
2 rows in set

SQL_DB >
```

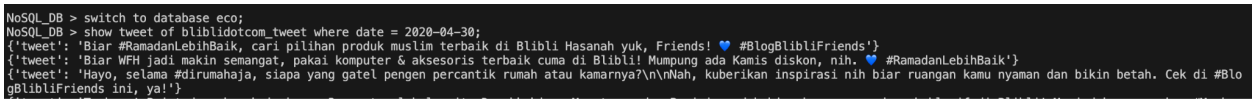
As mentioned above, the result of join, sort, and group operations will be stored in the *_running* folder. To display the result, we need to use the Retrieving Data query.
To show the connected table, run the command *"show all of teams_connect_ranking_running;"*.



Due to space limitations and for clearer demonstration purposes, some of the screenshots of the NoSQL implementation showcase the system using a smaller test dataset.

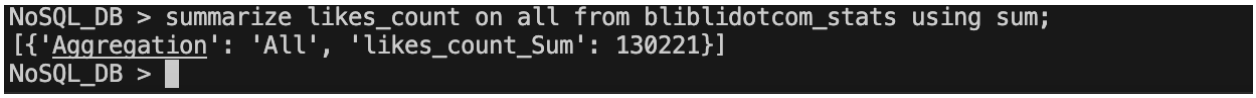
NoSQL System: Retrieving Data (Projection & Filtering)

To show the collection “bliblidotcom_tweet” with projection and filtering, run the command “show tweet of bliblidotcom_tweet where date = 2020-04-30;”.



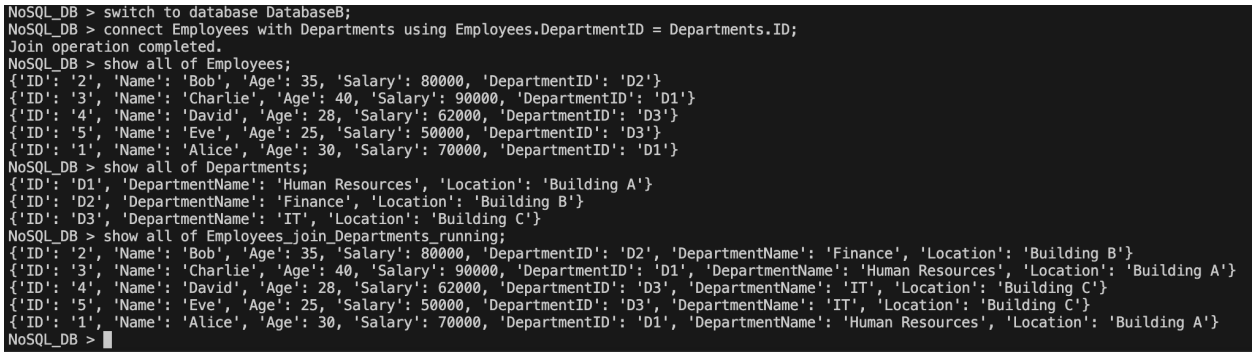
NoSQL System: Grouping and Aggregation

To get the sum of the likes_count field of the collection “bliblidotcom_stats”, run the command “summarize likes_count on all from bliblidotcom_stats using sum;”.



NoSQL System: Connecting Collections (Nested Loop Join)

To connect two collections based on a common feature, run the command “connect Employees with Departments using Employees.DepartmentID = Departments.ID;” and “show all of Employees_join_Departments_running;”



NoSQL System: Sorting Data (External Merge Sort)

To sort the collection “Employees”, run the command *“sort Employees by ID in asc order;”*.

```
NoSQL_DB > show all of Employees;
{'ID': '2', 'Name': 'Bob', 'Age': 35, 'Salary': 80000, 'DepartmentID': 'D2'}
{'ID': '3', 'Name': 'Charlie', 'Age': 40, 'Salary': 90000, 'DepartmentID': 'D1'}
{'ID': '4', 'Name': 'David', 'Age': 28, 'Salary': 62000, 'DepartmentID': 'D3'}
{'ID': '5', 'Name': 'Eve', 'Age': 25, 'Salary': 50000, 'DepartmentID': 'D3'}
{'ID': '1', 'Name': 'Alice', 'Age': 30, 'Salary': 70000, 'DepartmentID': 'D1'}
NoSQL_DB > sort Employees by ID in asc order;
NoSQL_DB > show all of Employees_ID_asc_running;
{'ID': '1', 'Name': 'Alice', 'Age': 30, 'Salary': 70000, 'DepartmentID': 'D1'}
{'ID': '2', 'Name': 'Bob', 'Age': 35, 'Salary': 80000, 'DepartmentID': 'D2'}
{'ID': '3', 'Name': 'Charlie', 'Age': 40, 'Salary': 90000, 'DepartmentID': 'D1'}
{'ID': '4', 'Name': 'David', 'Age': 28, 'Salary': 62000, 'DepartmentID': 'D3'}
{'ID': '5', 'Name': 'Eve', 'Age': 25, 'Salary': 50000, 'DepartmentID': 'D3'}
NoSQL_DB >
```

Learning Outcomes (Challenges Faced)

We encountered several obstacles, including devising methods to read data in chunks, adapting queries for CSV and JSON file formats, and implementing functions for join and sort operations. Through research, we found several methods for loading datasets without loading the entire database into the memory. We can read the data in chunks or use generators to read one row of data at a time. There is also a method involving Memory-Mapped Files. This method allows us to access the file as if it were in memory, but the OS manages the paging of data in and out of memory. In our search for a large JSON dataset, we found that the most extensive JSON datasets are system logs, typically containing one JSON object per line. Since we needed to design the system from scratch and had control over the database format, we opted to load data in chunks, reading a set number of lines into the main memory.

Moreover, since we are not allowed to the entire table or collection into the memory, we find it can be a bit tricky to join two CSV files or two JSON files together. We noted limited options for executing joins on JSON files. An example in MongoDB suggested a method similar to an outer left join on JSON files. After Professor Wu's lecture on query execution, we realized that a Nested Loop Join could be used to join two CSV/JSON files together. Furthermore, based on Professor Wu's lecture, we used external merge sort in the sort query. Since we already have *_running* folders to store the runtime files, we use *_running* folder to store sorted chunks and delete chunk files after the merge.

We also spend a considerable amount of time on the testing of queries. We initially missed some functionalities of a query that we had to fix. For example, we can only aggregate values based on the group in the Grouping and Aggregation function. We could not aggregate values without grouping. After finding out our design flaws, we changed the query language, added the statement "all" to refer to ungrouping, and changed the query function respectively.

Individual Contribution

Xuesong Shen
Graduate Major: Computer Science
Undergraduate Major: Computer Science
Skills: Python, SQL, NoSQL
Contribution:

- Project Proposal
- Project Midterm Report
- Design the query language
- Design system structure
- Design and implement the relational database system
- Final Report

Haoze Zhu
Graduate Major: Applied Data Science
Undergraduate Major: Statistics and Data Science
Skills: Python, R
Contribution:

- Project Proposal
- Project Midterm Report
- Design the query language
- Design and implement the NoSQL database system
- Final Report

Conclusion

In conclusion, this project has been a journey of exploration and learning in database systems. We successfully implemented two distinct database systems, one for SQL and the other for NoSQL databases, unified under a single command interface and a common query language. This project not only required us to delve deep into the intricacies of database management but also challenged us to create a query language capable of supporting a wide array of operations such as Projection, Filtering, Join, Grouping, Aggregation, Ordering, Inserting, Deleting, and Updating.

The knowledge and skills we have gained through this project are invaluable. We have not only honed our technical abilities in database management but also developed a deeper understanding of the complexities and nuances involved in designing and implementing database

systems. This project has set a strong foundation for our future endeavors in the field of data science and database management.

Future Scope

As we look forward, the potential for further development and enhancement of our database systems is vast. Our design for limiting memory usage is unstable depending on the database. We opted to load data in chunks by reading a set number of lines into the main memory. We limit the number of lines reads each time to control memory usage. However, the length of one line varies in different files, especially in JSON format. To improve memory usage, we need to find a more precise method to measure and monitor data reading.

Some query functions can also be improved to reduce query running time. Nested Loop Join, for instance, can impose a large I/O cost on the system, particularly with large datasets. For example, joining two collections in the NoSQL database eco could take minutes to run. In addition, if the dataset is too large, sorting could create an excessive number of sorted_chunk files in the _running folder, potentially leading to system breakdown. We need to find advanced algorithms to expand the system's scalability.

In summary, we envision incorporating more advanced features, improving user interface and interaction, and expanding the system's scalability and adaptability to various data formats and larger datasets.