DSCI 551 Project Final Report
Ziyu Chen

OurDb: a SQL-like, Scalable, and Flexible Metadata-Driven Database Design

Important Notices:
Google Drive link to all files:
https://drive.google.com/drive/folders/1Gd-D7-dQVrt7T5CBqW4vd5BkAp_JWtGT?usp=sharing

File 0: the final report, see DSCI551_project_final_report_Ziyu_Chen.pdf
File 1: OurDb Handbook, see *OurDb_Handbook.pdf*
File 2: Sample test cases, see *ourdb_sample_test_codes.txt*
File 3: OurDb program code, see *ourdb.py*

Introduction

The need for efficient data handling and processing in the era of big data is more crucial than ever. OurDB was developed as a solution to this need, offering a system that not only manages large volumes of data but also provides flexibility and ease of use. OurDB is a custom-designed database management system based on Python crafted to efficiently handle large datasets through a unique approach of managing data across multiple files, guided by a central metadata file. This system stands out for its user-friendly interactive command-line interface and its ability to perform a wide range of data operations, from basic CRUD (Create, Read, Update, Delete) functionalities to complex data manipulations like SELECT, WHERE, GROUP BY, or JOIN operations.

System Overview

- **Metadata-Driven Architecture**: At the heart of OurDB is a metadata file that maintains information about each table, including its name, location (number of files), headers (column names), current row count, total file size, and timestamps for creation and modification.

- **Data Storage and Management**: OurDB stores data across multiple files, avoiding the loading of entire datasets into memory, thus optimizing performance for large-scale operations.

- **Interactive Command-Line Interface**: Users interact with the database through a command-line interface, issuing commands for various operations.

Database Design

The basic idea is to read the large csv file by chunk and split the large file into several small files, each file contains at most 10000 lines. Program will store the table data into several csv files, each named table_name_0.csv, table_name_1.csv, until all the entries are written. By this design, when doing operations, we can avoid read the whole dataset for each query, instead, we can do tasks in queue and combine the results. In some cases, there are even no need to search for all csv files, e.g., when selecting the first few rows.

The indexing file for the database is named *tables_metadata.csv*. Its columns are: table_name, locations, header, current_row, total_file_size, create_time, modified_time, where table_name is the name for the table; locations means how many files are needed to store the entire dataset, it can become larger if we keep adding data into the table. Current_row saves the number of rows in the table_name_locations.csv file, if it is $\geq$ 10000, we need to write into a new file named table_name_(location+1).csv and set the current_row to 2 (since each file has a header). Header saves the header information for each table for better reference.

The program will read commands from user. The command will firstly go through several completed if statement check to determine what should be done. When deciding the basic usage, the program will call the related function, passing the command. Then, the function, with regular expression detection mechanism inside, will recognize what the user wants to do with the dataset, and call other functions to finish the tasks.

Regular expression plays an important role in recognizing user command. For instance, if a user inputs UPDATE TABLE employees SET age = 30 WHERE id = '123', the corresponding RE pattern would be something like r"UPDATE TABLE (\w+) SET (\w+) = (\d+) WHERE (\w+) = '(\w+)'". This pattern will correctly parse the table name (employees), column to update (age), new value (30), condition column (id), and condition value (123).

Pandas package serves as important role in handling data processing. Pandas' capabilities in data manipulation, aggregation, and its ability to handle large datasets efficiently help the developers of the database save a lot of time on implementing the basic selection, sorting, aggregation tasks. By using chunk processing as its code part, this program builds a simple database that can complete basic tasks as a traditional SQL software.

To successfully execute a complex ‘FIND FROM... FOR EACH... WITH ORDER…’ command, which involves filtering, grouping, aggregation, and sorting, the program is deliberately structured. First, we parse the command by using regular expression to extract key components of the command, such as the target table, columns for aggregation, grouping, and ordering criteria. Next, we load data in chunks. We will read from each small csv file and process the file to get a partial result. This approach ensures memory efficiency. For each chunk we need to apply the following operations:

- Filtering: If a WHERE condition is present, filter the data using Pandas' boolean indexing.
- Grouping and Aggregation: Use Pandas' groupby and aggregation methods (like sum, count, min, max) on the specified column(s)

We save the intermediate results in pandas dataframe. Then combine those partial results. After applying the sorting based on the specified criteria, we finished the last step. Finally, we make sure the dataframe is well compiled and neatly formatted and show the result to user.

Query Language

We created our own query language to facilitate this database design, though most usages are based on what SQL offers, with a consideration to make it more natural language_like. Usages like 'FIND FROM' 'UPDATE TO TABLE' should be really intuitive to understand. We also design some requirement, for example, when adding new entry, the value should be specified inside the square bracket []. This requirement is for better regular expression recognition. More detailed usage examples can be found in the user handbook.

Program Flowchart Representation

1. System Initiation:
    - Start by run .py file.
    - Initialize system (load metadata, prepare file system).
2. Command Input Loop:
    - Prompt user for input ('OurDB >').
    - Read command.
3. Command Parsing and Execution:
    - Check if the command is 'EXIT'. If yes, end the program.
    - If 'HELP', display help text.
    - If 'CLEAR DATABASE', confirm with the user and clear database if confirmed.
    - If 'DELETE TABLE', parse the table name and delete the specified table.
    - If 'SHOW TABLES' or 'SHOW TABLES NAME', display relevant table information.
    - If 'CREATE TABLE', create a new table based on the specified schema and initial data.
    - If 'IMPORT FROM', import data from a given CSV file into a new table.
    - If 'ADD TO TABLE', insert new data into the specified table.
    - If 'REMOVE FROM TABLE', delete data from the table based on specified conditions.
    - if 'UPDATE TO TABLE', update the data with given conditions
    - If 'FIND', find and display data based on conditions and aggregations.
    - If 'JOIN', perform a join operation between two tables and save the result as a new table.
    - If an unrecognized command, display an error message.
4. Loop Back or Exit:
    - Return to the command input loop unless the 'EXIT' command was given.

Sample Dataset:

We used two datasets from Kaggle.com.
The used_car_sale.csv file is 138 MB large, columns include manufacturer, model, year, mileage, engine, transmission, drivetrain, fuel_type, mpg, exterior_color, …
The car_manufacturers.csv file is relatively small. It contains information on the car manufacturer company. The columns include name, twitter, facebook, … This second table serves as a table to be joined. So, during the development span, we can better test the codes, with some meaningful grouping and joining operations.
The links to websites are:
Used Cars Dataset (kaggle.com)
Car Manufacturers (kaggle.com)

Future Directions:

- Enhancing the user interface for more intuitive interactions.
- Implementing additional features for data analysis and reporting.
- Improving algorithms to allow more completed compound queries.