USC University of Southern California

Cassandra Loh (USC ID: 5653-6631-08)

Project Link: DSCI 551: Final Project

Introduction & Overview

The objective of this project is to create either a relational or non-relational database system which is able to perform basic CRUD functionalities along with filtering, projection, joining and aggregating.

The code in the Google Drive titled "*script.py*" is my implementation of a basic command-line database system named "*MyDB*". MyDB is modeled after **non-relational databases** such as Firebase which provides real-time changes to the data stored. MyDB allows users to interact with the system through the command line interface and users are able to create tables, insert data, load data from files, and perform various operations such as finding records, picking specific fields that they are interested in, updating records, deleting records, sorting by a specific field, organizing by grouping and aggregating, and joining different tables. To mimic the functionality of a true database, these individual operations would be coded as functions, each with their own custom query language. The syntax of my query language takes inspiration from a little bit of natural language and also SQL-like queries for ease of parsing. MyDB stores data in JSON files in the same folder as the main .py script it runs on. The code includes exception handling for various operations, printing error messages when an exception occurs.

Description of Dataset

The dataset that I have chosen and showcased during the in-class demo is IMDB.json. This dataset was obtained from Kaggle and titled ***TV & Movie Metadata with Genres and Ratings***. At the point of pre-processing, the dataset is around 40MBs with 9 columns (title, genre, runtime, certificate, ratings, actors, description, votes and directors) and almost 130,000 rows of movie entries. Following data processing and cleaning which included steps such as list-wise deletion of entries with missing values, non-English characters and outlier values, I have transformed the remainder of the entries into a .JSON file. The final file size is about 8MB and the data is structured as follows:


```
{
    "Movie": "Mission: Impossible - Dead Reckoning Part One",
    "Genre": "Action, Adventure, Thriller          ",
    "Runtime": "163 min",
    "Certificate": "UA",
    "Rating": 8.0,
    "Stars": [
      "Tom Cruise, ",
      "Hayley Atwell, ",
      "Ving Rhames, ",
      "Simon Pegg",
      ""
```

```
    ],
    "Director": [
      "Christopher McQuarrie"
    ]
  },
  {
    "Movie": "Sound of Freedom",
    "Genre": "Action, Biography, Drama          ",
    "Runtime": "131 min",
    "Certificate": "PG-13",
    "Rating": 7.9,
    "Stars": [
      "Jim Caviezel, ",
      "Mira Sorvino, ",
      "Bill Camp, ",
      "Cristal Aparicio",
      ""
    ],
    "Director": [
      "Alejandro Monteverde"
}
```


Implementation:

Functionalities

Classes:

- **MyDB Class**
  - The MyDB class is derived from the Cmd class, which provides a framework for building command line oriented interpreters. It contains all the functions for database operations.

Functions & Query Syntax:

- **Constructor ("__init__") -** This code is part of the MyDB class and it is responsible for initializing the instance variables when a new MyDB object is created.
  - **super().__init__()** - This line calls the constructor of the parent class, 'Cmd'. The 'Cmd' class is part of the Python Standard Library and is used for building line-oriented command interpreters. By calling the "__init__" method of the parent class, the MyDB class inherits the functionality of the 'Cmd' class.

USC University of Southern California

Cassandra Loh (USC ID: 5653-6631-08)

- **self.prompt = 'MyDB' >** - This line sets the command prompt for the interactive shell. The prompt is the text that appears before the user input a command. In this case, I have set the prompt to 'MyDB >', indicating that the user is interacting with the MyDB system and their input should follow after the prompt.
- **self.table = None** - This line initializes the self.table instance variable to None. The self.table variable is used to keep track of the currently selected table in the database. Initially, when the MyDB object is created, no table will be selected.
- **self.data = []** - This line initializes the self.data instance variable as an empty list. The self.data variable is used to store the data of the currently selected table when loaded in on a line by line basis. It will hold the records and fields of the table during the session. Initially, when the MyDB object is created, no data is loaded so it is initialized to an empty list.

➢ **Create Tables [do_create (self, line)]**
- This function creates a new table with the given name.
- The function also includes an error handling to check if the table name already exists before creating a new one.
- Command Syntax: create <table_name>

➢ **Inserting Data [do_insert (self, line)]**
- This function inserts data into the currently selected table.
- The code parses the input JSON string and appends the data to the selected table's JSON file.
- Command Syntax: insert {key: value, key: value, etc..}

➢ **Load Data [do_load (self, table)]**
- Loads the data from an existing table into the current session to be worked on.
- It reads all the data from the specified table's JSON file and appends them to the initial empty list in the constructor function to be worked on in main memory.
- Command Syntax: load <table_name>

➢ **Find Records [do_find (self, line)]**
- This function searches for records in the currently selected table based on a specified condition and returns all the information for that record.
- It supports basic querying using regular expressions.
- Command Syntax: Find record whose <key> is <value>

➢ **Pick Fields in Records [do_pick (self, line)]**
- This function selects specific fields from records in the currently selected table based on a specified condition. It supports filtering using a WHERE clause.
- Command Syntax: pick <key>, <key> WHERE <key> at least/is/greater than/less than <value>

➢ **Update Records [do_update (self, line)]**
- This function updates records in the currently selected table based on a specified condition.
- It supports updating a specific field with a new value.
- Command Syntax: update <key> = <new_value> WHERE <key> = <value>

USC University of Southern California

Cassandra Loh (USC ID: 5653-6631-08)

- ➢ **Delete Records [do_delete (self, line)]**
  - This function deletes records from the currently selected table based on a specified condition. It supports deleting records using various operators.
  - Command Syntax: delete <key> = <value>
- ➢ **Sorting Records [do_sortby (self, line)]**
  - This function allows us to sort the records in the currently selected table by a specified field in an order (ascending or descending).
  - Command Syntax: sortby <key> ASC/DESC
- ➢ **Groups and Aggregate Records [do_organize (self, line)]**
  - This function allows users to group or aggregate records in the currently selected table based on specified fields and aggregate functions (SUM, AVG, COUNT).
  - Command Syntax: organizeby <key> <key> SUM/AVG/COUNT
- ➢ **Join Records [do_join (self, line)]**
  - This function allows users to join two tables based on a specified common field.
  - Command Syntax: join <table> <table> <key>
- ➢ **Exit [do_exit (self, line)]**
  - This function allows the user to exit the database's interactive shell once they are done using it.
  - Command Syntax: exit()

Main Execution:

- ➢ The "if __name__ == '__main__': " block creates an instance of the MyDB class and starts the command-line loop using "cmdloop()".

Tech Stack

- **Programming Language**
  - Python
- **File Handling Format**
  - JSON for persistent storage
- **Command-Line Interface Capabilities**
  - "Cmd" module for building an interactive CLI
- **Regular Expressions**
  - Used to parse and validate user input for queries.
- **Data Manipulation and Analysis**
  - Standard Python Libraries:
    - JSON
    - re
    - Itertools
    - Operator
- **Error Handling**

USC University of Southern California

Cassandra Loh (USC ID: 5653-6631-08)

- Individual exception handling in each function for robustness to catch wrong query syntax or input.

Screenshots

- Create Function

```
def do_create(self, line):
    try:
        self.table = line.strip()
        if os.path.exists(f'{self.table}.json'):
            print("Table already exists.")
            return

        with open(f'{self.table}.json', 'w') as f:
            json.dump([], f)
        print(f"Table {self.table} created.")
    except Exception as e:
        print(f"Error creating table: {e}")
```

- Join Function

```
def do_join(self, line):
    try:
        table1, table2, on = line.split()
        with open(f'{table1}.json', 'r') as f1, open(f'{table2}.json', 'r') as f2:
            data1 = json.load(f1)
            data2 = json.load(f2)

        joined_data = [
            {**d1, **d2} for d1 in data1 for d2 in data2 if d1[on] == d2[on]
        ]

        for item in joined_data:
            print(json.dumps(item, indent=2))

    except Exception as e:
        print(f"Error executing join: {e}")
```

USC University of Southern California

Cassandra Loh (USC ID: 5653-6631-08)

- Find Function

```
def do_find(self, line):
    if self.table is None:
        print("No table is loaded/selected.")
        return

    try:
        # Corrected regular expression pattern
        match = re.search(r"\w+ whose (\w+) is (.+)", line)
        if match:
            field, value = match.groups()
            value = value.strip()

            # Check if the first record's field value is a list
            results = []
            if isinstance(self.data[0][field], list):
                try:
                    for record in self.data:
                        for item in record[field]:
                            if value in str(item) and record not in results:
                                results.append(record)
                except Exception as e:
                    pass
            else:
                try:
                    results = [record for record in self.data if value in str(record[field])]
                except Exception as e:
                    pass

            if len(results) == 0:
                print("No matching records found.")
                return

            for record in results:
                print(json.dumps(record, indent=2))
        else:
            print("Invalid query format.")
            return

    except Exception as e:
        print(f"Error executing find: {e}")
```

Learning Outcomes

- Database Design Adaptation
    - Challenge: In my project proposal, the initial database I envisioned was a relational database architecture for managing data sourced from the IMDB.csv file on Kaggle. However, due to a significant reduction in the dataset size after data cleaning, a strategic decision was made to transition to a non-relational database model. To achieve this, I transformed the remaining rows from IMDB.csv to a JSON key and value pair format to align with the adapted database design.
- Complex Query Parsing
    - Challenge: The implementation of a custom query language involved a lot of parsing of user queries to resemble a natural language processing (NLP) scenario. Handling the intricate query structures, akin to an advanced query language, posed a significant challenge as I've never worked with this level of parsing with regular expressions. To leverage regular expressions for query interpretation required an in-depth understanding

of parsing mechanisms and involved a lot of trial and error before the final syntax was achieved. ‘Ve learned that it is definitely easier to start with the queries and work backwards to parse it.

- Dependency Limitations and Functional Equivalency
    - Challenge: The project encountered limitations in utilizing established Python packages such as pandas. These are commonly employed for streamlined data manipulation in database-like scenarios. However, since we were not allowed to use quite a few of the packages, the challenge involved was to reproduce these functionalities of a robust database management system within the constraints of the available Python packages or to manually hard code it without the help of the packages.

Conclusion

In conclusion, the developed code represents a command-line interface (CLI) for managing tabular data stored in JSON files, simulating basic database functionalities. The implementation encompassess features such as creating tables, inserting and manipulating data, and conducting data analysis operations. Several challenges were encountered along the way and addressed throughout the development of the project, contributing to a deeper understanding of database design, query parsing, and the creation of custom functionalities without the help of external packages in the Python environment.

This project served as a valuable learning experience, pushing the boundaries of database management within a Python environment. The challenges encountered highlighted the importance of adaptability, creativity, and a deep understanding of both the standard libraries and Python’s external packages in developing functional and efficient solutions. It greatly enhanced my appreciation for the existing databases and the work that went into it to get it where it’s at now.

Future Scope

This project serves as a foundation for a basic CLI for managing tabular data stored in JSON files. Some potential future enhancements that can be achieved:

1. Security Features
    a. Add user authentication and authorization mechanisms to secure access to the database.
    b. To handle sensitive data, a layer of encryption can be implemented/
2. Data Validation
    a. Enhance data validation to ensure the integrity and consistency of stored data.
    b. Implement constraints such as unique ID, foreign keys and data type validations.
3. Query Language Expansions
    a. The current query language has a very basic structure. Perhaps in the future, it can be extended to support a broader range of commands for complex queries.
    b. Implement more advanced filtering, grouping and aggregate functions so that it can be used for data analysis.