DSCI 551 Final Project Report

Qingyi Feng & Chuqi(Angel) Jin

2025 Spring - ChatDB17

May 9th, 2025

Table of Contents

Introduction

As database systems are widely deployed in various applications, users demand higher intelligence and convenience in interaction methods. Traditional query methods rely on SQL or specific NoSQL commands, which have a high threshold for users with non-technical background. To increase database accessibility and user experience, this project planned to develop a natural language interface (NLI) for both RDBMS (MySQL) and NoSQL (MongoDB) databases using a real-world large language model (LLM).

By integrating an LLM, the system allows users to speak directly to the database in natural language, such as "What listings allow pets?" to "Show houses in Florida under $2,000." The system will then automatically recognize the user's intent, identify the target database, extract the key entities, and translate the inputs into corresponding SQL or MongoDB query commands to get the requested data, before returning the results to the user.

In addition, we want this NLI to offer not only basic query operations, but also database modification functions like insertion, update, and deletion, as well as preliminary CRUD full-flow capabilities. At the same time, we created a context-aware technique that allows users to ask questions continuously while the system automatically retains the prior round of query conditions, resulting in a more natural "multi-round conversational data exploration".

Planned Implementation

In the early stage of the project, we have developed a detailed implementation plan to ensure the functional completeness and development feasibility of the system. Considering that this course project requires handling both relational and non-relational databases, we chose to use MySQL as the relational database and MongoDB as the non-relational database to support the storage and manipulation of structured and semi-structured data, respectively. To bridge the natural language-database interaction, we planned to introduce an advanced large-scale language models (LLMs) Google Gemini 2.0 flash, for parsing the user's natural language inputs and automatically generating the corresponding SQL or MongoDB query statements. This large-scale language model has powerful language understanding and generation capabilities, and can generate structured query language (SQL) or MongoDB commands based on natural language questions input by users.

Moreover, we planned to implement this NLI on our real-world dataset found on Kaggle, a real apartment rental listings dataset, which contains about 100,000 records and 22 attributes covering multi-dimensional information such as location, price, size, amenities, pet policy, and so on. In the following process, we plan to clean the raw data, remove features that contain too many null values and redundancy, make sure all listings are unique in this dataset, and filter out listings by regions, keeping only data from 5 east coast states in the U.S. (Virginia, North Carolina, Florida, Maryland and Massachusetts) in order to control the data size and optimize the processing efficiency.

In terms of database structure design, we planned to store 3–4 tables in each database. For MySQL, we created a database named ‘aptadditional’, which stores additional information about each apartment listing. The tables include:

- **pricing (id, price, currency)**: Primary key: id. Stores the listing price and the currency. This table serves as the anchor for other tables in MySQL.
- **price_details (id, price_display, price_type)**: Primary key: id; foreign key: id. Contains the actual displayed price along with the payment type (weekly, monthly, or yearly).
- **amenities (amenity_id, amenity_name)**: Primary key: amenity_id. Stores amenity IDs and their corresponding names.
- **property_amenities (id, amenity_id)**: Primary key: (id, amenity_id); both are also foreign keys. Links each apartment listing to its available amenities.
- **pets(id, pets_allowed, fee)**: Primary key: id; foreign key: id. Contains the pet policy for each listing and indicates whether an additional fee is required.

In terms of relationships, pricing and price_details have a one-to-one relationship, which is implemented in our database using a shared primary key. In more detail, pricing.id is the pricing table's primary key, whereas price_details.id is both the primary and foreign key that references pricing.id. There is also a many-to-many relationship between amenities and the property_amenities table, as each apartment listing may contain several amenities, and each amenities type may be shared by numerous apartment listings. Furthermore, there is a one-to-one relationship between the pets table and the pricing table; the listing id serves as both the primary and foreign key for the pets tables that reference the pricing.id. Additionally, all foreign keys are enforced with constraints(on delete cascade on update cascade) to make sure that updates and deletions are reflected across related tables in real time.

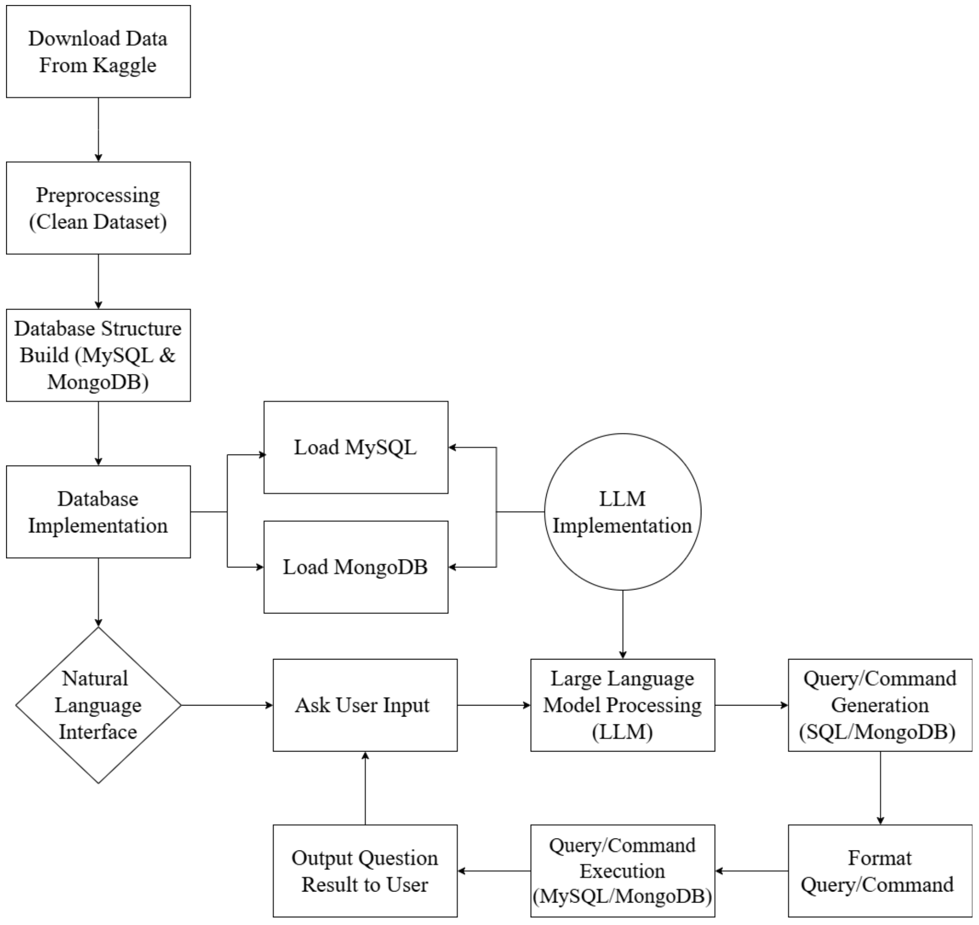
For MongoDB, we designed a database named ‘rental’, which stores general and structural information about apartment listings in a more flexible, document-oriented format. The collections include:

- **general_info (id, title, body)**: contains the listing ID, title, and description text of each apartment.
- **location (id, cityname, state, latitude, longitude)**: includes city name, state, latitude, and longitude.
- **property_details (id, square_feet, bedrooms, bathrooms)**: stores square footage, number of bedrooms, and bathrooms.
- **media (id, has_photo)**: indicates whether each listing includes photos.
- **sources (id, source, time)**: shows the listing website and the post timestamp.

These designs not only clarify the data logic, but also ease later modeling and querying in SQL and NoSQL databases, respectively. The MySQL and MongoDB table/collection structures will be constructed in accordance with the definitions provided above, and data will be inserted and retrieved using Python scripts. We then began the LLM implementation. To get the Gemini-2.0 flash model to work in our NLI, we generated API keys on Google. Then we utilized a JSON file to store our API key for data privacy and imported it into our NLI. Following that, we begin writing our prompt to set up the Gemini model by connecting to our loaded databases using MySQL and MongoDB. We briefed Gemini about our database structure at the customize prompt, including which tables/collections were stored in MySQL and MongoDB, as well as the features (just feature names) that were saved in each table/collection.

In addition, we intended to allow natural language-driven data manipulation in the system, including basic CRUD (add, delete, modify, and lookup) operations. As a result, in the customized prompt, we additionally customize the LLM model to only return specific queries/commands in order for our NLI to function properly. We anticipate that our final NLI will provide two types of core functions: query tasks (data analytics, filtering, and searching) and data manipulation tasks (inserting, deleting, and updating records). To improve the system's interactivity, we also intend to create a user interactive feature in the Jupyter Notebook that mimics the function of a frontend UI. Our entire development process is divided into phases, such as data cleaning, database modeling, LLM integration, query function implementation, and system optimization, each with its own timeline, to ensure that each stage is tightly integrated and that project goals are continuously fulfilled.

Architecture Design


Download Data From Kaggle
Preprocessing (Clean Dataset)
Database Structure Build (MySQL & MongoDB)
Database Implementation
Load MySQL
Load MongoDB
LLM Implementation
Natural Language Interface
Ask User Input
Large Language Model Processing (LLM)
Query/Command Generation (SQL/MongoDB)
Format Query/Command
Query/Command Execution (MySQL/MongoDB)
Output Question Result to User

**Flow Diagram*

- Stage 0: Pre-NLI

Regarding the data cleaning, database structure and LLM implementation, we explained in the previous section. For the database implementation, we used python files to import and load all listings into MySQL and MongoDB databases. We connected to the MySQL database using ‘pymysql’ and used cursor.execute to create a MySQL structure. Then, load each table data directly from its corresponding CSVs using ‘sqlalchemy’ into each table. We also used Python to import the cleaned listing data into MongoDB, connected to the database through the ‘pymongo’ library and insert each record into the specified collection after the fields and wrapping the structure. The tables in each collection are related by an ID field, forming logical one-to-one or one-to-many relationships.

- Stage 1: Ask User Input

In the beginning, users will see a tiny welcome section that says: "Welcome to the Unified Database Natural Language Interface!" followed by another line: "Type 'exit' to quit". These two lines serve as instructions for users to read before interacting with our interface. Then, to ensure obvious structure, we utilized a divider line to separate the instructions from the actual interactive part below them. Inside the interactive section, users will see a command line that asks them to enter their input as "Enter your question" followed by a rectangle box where users can freely enter their question in natural language such as "Show me apartments in Florida under $2000" or "Which Los Angeles listings allow pets?" Furthermore, if the user has previously used our system, by pressing the up and down buttons, they can retrieve their previous questions with a single click.

- Stage 2: Large Language Model (LLM) Processing

After the users enter their requests, the system will proceed to the inner processing phase by first passing through LLM, which performs intent recognition, keyword extraction, entity recognition, and contextual understanding to capture the core semantics of the user's questions. Based on the information provided in our customized prompt, it will automatically determine which database queries/commands to generate in order to answer user questions. After the LLM has made their decision, it will display a confirmation message in our user interactive section that says: "Engine Selected: mysql or mongodb". This message confirms that the LLM successfully interpreted the user questions and selected the corresponding database, which is ready to generate the query/commands. However, if the LLM fails to answer or determine the database, an error message will be displayed based on our try and except structure in our code to inform users that the LLM did not properly process the input questions.

- Stage 3: Query/Command Generation (SQL/MongoDB)

After determining the database, the system will generate corresponding SQL query or MongoDB commands based on our custom prompt requirement. In the requirement, we regulate the LLM to generate either a single line SQL or a single line PyMongo command. For SQL queries, always use full table names to reduce mistakes. For MongoDB commands, because we will use pymongo to access our database, regular mongodb commands can not directly be used to access our database, therefore, we mentioned that in our prompt to LLM to generate pymongo commands instead. Specifically, generate find commands for simple requests, and use aggregate for advanced operations with '$match', '$group', '$sort', '$limit', '$skip', '$project', and '$lookup'. Moreover, we let LLM decide when to use '.insert_one', '.insert_many', '.update_one', '.update_many', '.delete_one', '.delete_many' for data modification. More importantly, do not use '.count_documents()', '.find_one()', etc. Because those will not work on our database access. Also, for the join function, if the requested fields are from different collections, join them based on a common key. Finally, return all responses in a single-line JSON and with no further explanation needed to avoid LLM generating extra information.

- Stage 4: Format Query/Command
To correctly access data from created LLM queries/commands, we processed the raw LLM result before displaying it to the user and accessing the databases. By detecting and parsing the single-line json, if the required query/commands are wrapped in formatting tags such as '```json' and '```', we will remove them and load the json; if the system cannot identify the single-json line, it will display an error message to notify the user of the parsing issue. Then, before utilizing the queries/commands to access the database, we perform format-specific handling. To avoid formatting issues in SQL queries, we use '%%' instead of '%'. For PyMongo commands, we replace '```python' and '```python' with an empty space and remove any additional whitespace. After formatting, the SQL query/Pymongo Commands are ready to run on both the MySQL and MongoDB databases. The formatted query/commands will also output to user interact section as 'Generated Query: query/commands'

- Stage 5: Query/Command Execution (MySQL/MongoDB)
At this point, the generated query/commands will be handed to the appropriate database engine for execution. To retrieve or modify data, the system connects to both the local MySQL instance (aptadditional database) and the MongoDB instance (rental database). Furthermore, if an error occurs as a result of query/command execution, an error message will appear to advise the user of the specific database with which it is having difficulties or if it is a general execution error.

- Stage 6: Output Question Result to User
Finally, the database returns results that are prepared for visual presentation. MySQL results will be displayed in table format, whereas MongoDB results will be displayed as dictionaries. After displaying the current question's result, it will automatically return to stage 1 and display a new

command line for the user to enter new questions. The user may choose to leave by entering "exit" or continue asking questions.

Implementation

1. Functionalities

This system implements a Natural Language interface(NLI) based on the Large Language Model(LLM) to support users to access and operate two types of database systems in natural language: relational database(RDBMS: MySQL) and non relational database(MongoDB). The system supports a variety of core functions, including structured data retrieval, aggregation and analysis, sorting and filtering, data modification(e.g. Insert, update and delete), and multiple rounds of dialogues, which comprehensively improves the intuitiveness, flexibility, and intelligence of data access. The following are the main functional modules implemented in this project:

1. **Natural Language Question Input**: Users can ask questions in natural language, and LLM will automatically turn them into SQL or MongoDB query statements.
2. **Gemini 2.0 Flash LLM**: After users ask a question, the system sends it to LLM for interpretation and query generation.
3. **Display Generated Queries/Commands**: After going through LLM, the system will present two information sections above the output results: the selected database and the created query/commands, which will be needed in the following stages.
4. **Query/Commands Execution**: The system automatically executes the generated SQL query or PyMongo command on the appropriate database.
5. **Schemas & Data Explorations**: Users can explore MySQL and MongoDB databases to discover their structure, tables/collections, attributes, and extract sample data.
6. **MySQL Queries**: To answer users' questions, the LLM will construct typical queries such as SELECT, FROM, JOIN, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, and OFFSET.
7. **MySQL Data Modification**: The system provides natural language data management operations such as inserting, deleting, and updating database records.
8. **MongoDB Commands:** To answer questions from users, the LLM will generate PyMongo commands such as find, projection, aggregate with $match, $group, $sort, $limit, $skip, $project, and join with $lookup.
9. **MongoDB Data Modification**: The system allows users to modify data using natural language. The generated commands include insertOne, insertMany, deleteOne, deleteMany, updateOne, and updateMany.
10. **Query/Command Result Output**: Whether utilizing SQL or MongoDB queries, the system standardizes the results and presents them to the user in a clear and readable format.

11. **Error Handling Mechanism**: When the user enters ambiguous information, misspells a field, or the model is not successfully parsed, the system will display an error message or the default explanation and prompt the user to retry.

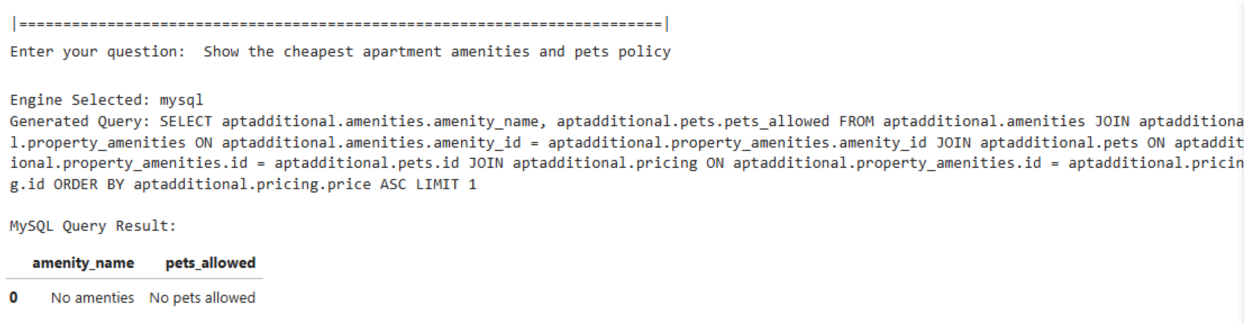
Overall, this system achieves complete functionality in natural language query parsing, structured retrieval, statistical analysis, data editing, and result presentation. Users can access, manipulate and analyze data efficiently through natural language, which significantly improves the usability and user experience of the database system and provides strong support for the integration of natural language and data interaction.

2. Tech Stack

- Language: Python 3.12.4
  - Libraries:
    - MySQL(pymysql, sqlalchemy)
    - MongoDB(pymongo)
    - pandas
    - json
    - os
- Databases
  - MongoDB
  - MySQL
- LLM: Google Gemini Flash 2.0 API
  - google-genai
- Hosting: Localhost
- Tools
  - VS Code
  - Jupyter Notebook

3. Implementation Screenshots

MySQL:

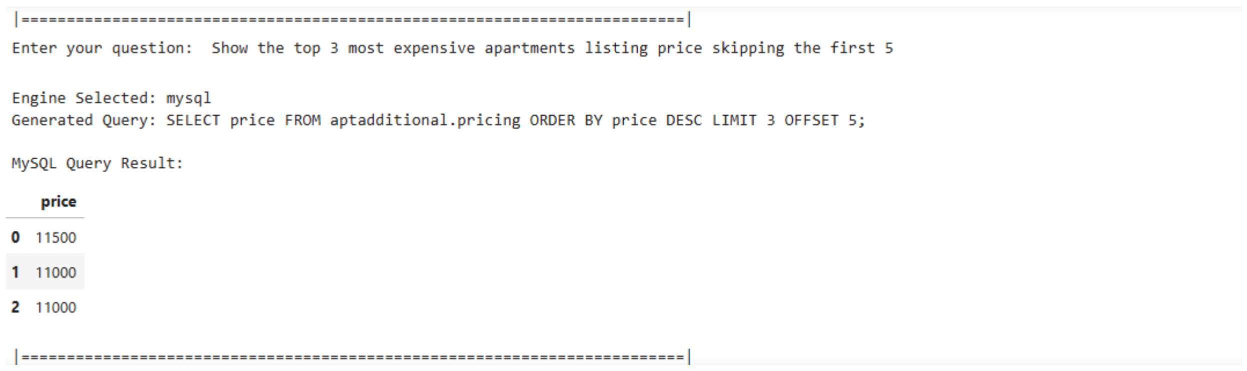


```
|========================================================================|
Enter your question:  Show the cheapest apartment amenities and pets policy


Engine Selected: mysql
Generated Query: SELECT aptadditional.amenities.amenity_name, aptadditional.pets.pets_allowed FROM aptadditional.amenities JOIN aptadditiona
l.property_amenities ON aptadditional.amenities.amenity_id = aptadditional.property_amenities.amenity_id JOIN aptadditional.pets ON aptaddit
ional.property_amenities.id = aptadditional.pets.id JOIN aptadditional.pricing ON aptadditional.property_amenities.id = aptadditional.pricin
g.id ORDER BY aptadditional.pricing.price ASC LIMIT 1

MySQL Query Result:
```

	amenity_name	pets_allowed
0	No amenties	No pets allowed



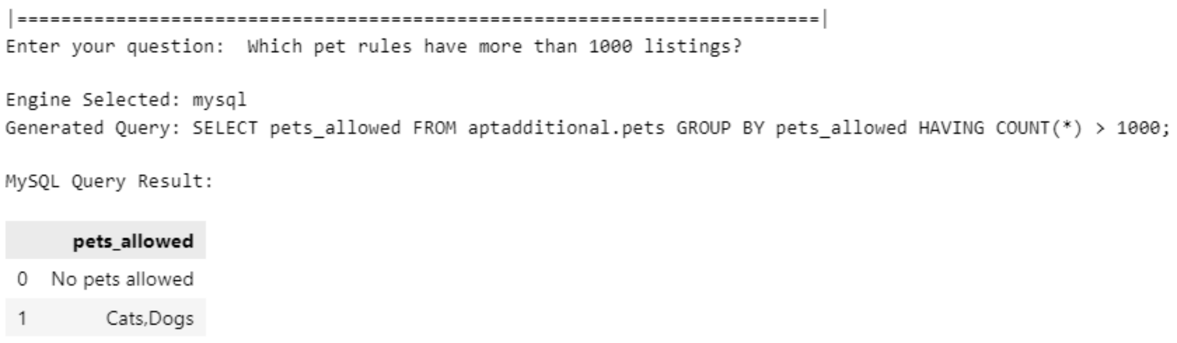
```
|========================================================================|
Enter your question:  Show the top 3 most expensive apartments listing price skipping the first 5

Engine Selected: mysql
Generated Query: SELECT price FROM aptadditional.pricing ORDER BY price DESC LIMIT 3 OFFSET 5;

MySQL Query Result:
```

	price
0	11500
1	11000
2	11000

```
|========================================================================|
```

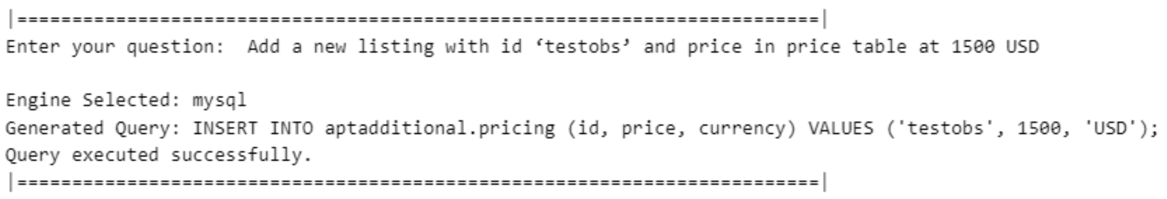


```
|========================================================================|
Enter your question:  Which pet rules have more than 1000 listings?

Engine Selected: mysql
Generated Query: SELECT pets_allowed FROM aptadditional.pets GROUP BY pets_allowed HAVING COUNT(*) > 1000;

MySQL Query Result:
```

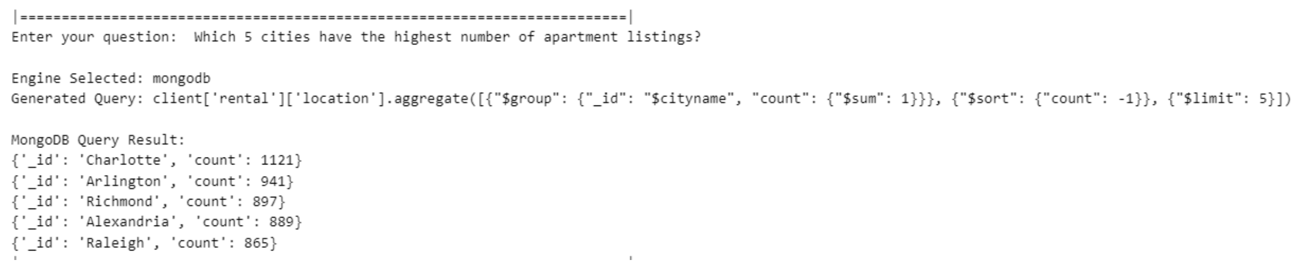
	pets_allowed
0	No pets allowed
1	Cats,Dogs



```
|========================================================================|
Enter your question:  Add a new listing with id ‘testobs’ and price in price table at 1500 USD

Engine Selected: mysql
Generated Query: INSERT INTO aptadditional.pricing (id, price, currency) VALUES ('testobs', 1500, 'USD');
Query executed successfully.
|========================================================================|
```

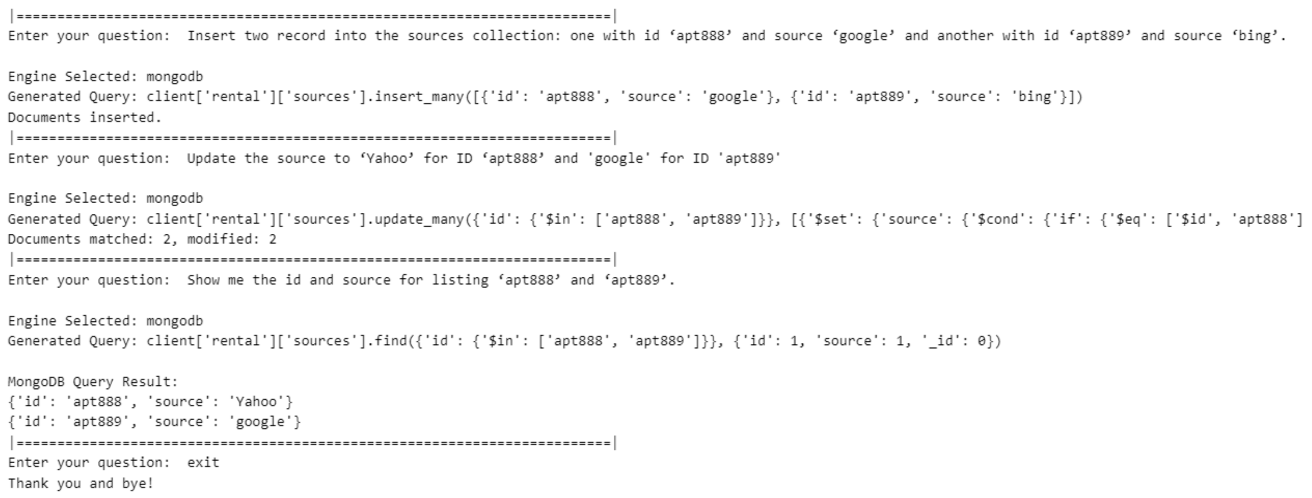
MongoDB



```
|========================================================================|
Enter your question:  Which 5 cities have the highest number of apartment listings?

Engine Selected: mongodb
Generated Query: client['rental']['location'].aggregate([{"$group": {"_id": "$cityname", "count": {"$sum": 1}}}, {"$sort": {"count": -1}}, {"$limit": 5}])

MongoDB Query Result:
{'_id': 'Charlotte', 'count': 1121}
{'_id': 'Arlington', 'count': 941}
{'_id': 'Richmond', 'count': 897}
{'_id': 'Alexandria', 'count': 889}
{'_id': 'Raleigh', 'count': 865}
```



```
|========================================================================|
Enter your question:  Insert two record into the sources collection: one with id ‘apt888’ and source ‘google’ and another with id ‘apt889’ and source ‘bing’.

Engine Selected: mongodb
Generated Query: client['rental']['sources'].insert_many([{'id': 'apt888', 'source': 'google'}, {'id': 'apt889', 'source': 'bing'}])
Documents inserted.
|========================================================================|
Enter your question:  Update the source to ‘Yahoo’ for ID ‘apt888’ and 'google' for ID 'apt889'

Engine Selected: mongodb
Generated Query: client['rental']['sources'].update_many({'id': {'$in': ['apt888', 'apt889']}}, [{'$set': {'source': {'$cond': {'if': {'$eq': ['$id', 'apt888']
Documents matched: 2, modified: 2
|========================================================================|
Enter your question:  Show me the id and source for listing ‘apt888’ and ‘apt889’.

Engine Selected: mongodb
Generated Query: client['rental']['sources'].find({'id': {'$in': ['apt888', 'apt889']}}, {'id': 1, 'source': 1, '_id': 0})

MongoDB Query Result:
{'id': 'apt888', 'source': 'Yahoo'}
{'id': 'apt889', 'source': 'google'}
|========================================================================|
Enter your question:  exit
Thank you and bye!
```

Learning Outcomes

This project has enabled us to gain in-depth understanding and competence in various aspects of database systems, natural language processing, and system integration. First of all, in terms of database modeling, we learned and mastered the use of relational databases (MySQL) and non-relational databases (MongoDB). By modeling the same batch of apartment rental data in two databases, we deepened our understanding of normalized design, multi-table associations, primary and foreign key relationships, and document-based structure, and realized how to choose the appropriate database structure under different query requirements. Secondly, in the data processing session, we improved our ability to handle real data. Facing the problems of null values, redundant fields, inconsistent formats and duplicate records in the raw data, we learned how to clean, transform and split the structure through Python (mainly using Pandas) to build a clean and usable structured dataset for subsequent database design and query.

The most challenging part was the development of the natural language processing module. We introduced the Google Gemini language model into a database application for the first time, and learned how to do prompt design (prompt word engineering), how to guide the model to understand structured semantics, and accurately map natural language to SQL or MongoDB query commands. In the process, we gained a deeper understanding of the capability boundaries and logical reasoning of language models. In addition, during the whole system integration and development process, we also improved our teamwork skills, including division of labor, code collaboration, debugging and testing, etc. We also learned a lot.

Overall, this project not only strengthened our technical skills, but also cultivated our ability to solve practical problems, face technical complexity, and collaborate in team development, laying a solid foundation for our future work in data-driven product development.

Challenges Faced

During the implementation of the project, although we accomplished the intended goals, we also faced many technical and implementation challenges. These problems not only tested our knowledge base and problem solving ability, but also prompted us to keep learning and adjusting the program in the real world. The first challenge we faced is the data filtering during the data cleansing phase. When we first started working with the raw dataset, we encountered a large number of null values, inconsistent field naming, duplicate IDs, and inconsistent time formats, especially in fields such as category and address. We had to decide which fields should be retained and which should be discarded, and ensure that the cleaned data structure would map consistently across MySQL and MongoDB. In the end, we figured out the way to clean and filtered data down to only 5 states to reduce the original dataset size for us to better work on.

Additionally, in our initial plan, we plan to implement all tables in both databases, and separate them into 3 different databases for implementation. However, after testing that on our NLI, we found that if we have the same structure for both databases, the LLM will always choose to generate MySQL queries instead of MongoDB commands, therefore the implementation will not be considered successful. Therefore, we decided to redesign our database structures for both databases to have a better performance for our NLI to work. The updated data structure is written in our above planned implementation section. After finishing building our database structure, we begin to load our data into MySQL and MongoDB. For MongoDB, we quickly figured out a way to load the entire dataset in and then distribute it to its corresponding collections. But for MySQL, we couldn't find a simple way to do the similar method. Therefore, what we did is to subset data into separate CSVs based on needs of each table and load each CSV directly into MySQL tables.

Moreover, another challenge we faced is the accuracy and contextual understanding of natural language parsing. In the process of using LLM(Google Gemini) to automatically generate SQL or MongoDB query statements, we found that the model sometimes fails to accurately understand the intent of user input, especially when encountering ambiguous expressions or in the presence of spelling errors. In addition, contextual understanding in consecutive conversations (e.g., “Show me listings in Boston” → “Which have 2 bedrooms?”) also suffers from insufficient model memory or ambiguity. To address these issues, we iteratively adjusted the prompts and tried to incorporate prompt templates, example statements, and additional instructions to improve the accuracy of model generation.

The final challenge is for the query result visualization. When query results are returned, especially when nested fields, multiple matches, or no results are involved, we need to standardize the format of the output so that users can clearly understand the query results. Although this part has nothing to do with model generation, it has a great impact on the user experience, and thus becomes one of the focuses of our continuous debugging and optimization. We implemented various results for different kinds of situations to output clear messages back to the user for each input and also used a line divider to separate results of each question more clearly and organized.

Individual Contribution

- Data Cleaning (Qingyi Feng & Chuqi(Angel) Jin)
- LLM Setup (Qingyi Feng & Chuqi(Angel) Jin)
- Database Design (Qingyi Feng & Chuqi(Angel) Jin)
- MySQL Database Building (Chuqi(Angel) Jin)
- MongoDB Database Building (Qingyi Feng)
- LLM Implementation (Qingyi Feng & Chuqi(Angel) Jin)
- Implementation (Qingyi Feng & Chuqi(Angel) Jin)
- Code debugging & Testing (Qingyi Feng & Chuqi(Angel) Jin)
- Project Report (Qingyi Feng & Chuqi(Angel) Jin)

Conclusion

This project maps natural language queries to structured database operations, combining the advantages of natural language and traditional database management systems. By using MySQL and MongoDB as dual back-end databases, and and introducing Google Gemini as the natural language processing engine, we have successfully implemented a natural language interface that can understand users' natural language inputs and perform corresponding operations on structured and semi-structured data.The project covers a complete engineering development process, starting from data cleaning and restructuring, step-by-step completion database modeling, language model integration, query generation and execution module development and finally system testing and performance optimization. In the system design, we have built a scalable data architecture, so the natural language interface can cross SQL and NoSQL environments, and support diversified data analysis scenarios and user interactions.

During the implementation of the project, we understood the technical challenges and implementation mechanisms behind the natural language interface, and comprehensively improved our capabilities in data management, system architecture design, multi-database integration, and collaborative team development. The final system has preliminary practical value and can be used as a prototype of an intelligent customer service platform, apartment listings search tool or natural language data analysis interface. Overall, this project not only helped us successfully apply the theoretical knowledge we learned in class to practice, but also laid a solid foundation for us to engage in intelligent, data driven application research and development in the future.

Future Scope

Although this project has implemented LLM to interpret natural language questions and created queries/commands for database interaction at this point, we acknowledge that there is still potential for system optimization and growth. In the future, we hope to enhance our NLI system in several key areas. First, we want to help our NLI process more ambiguous natural language questions, which will improve model interpretation and fault tolerance. We'd like to improve the system's error handling capability in the event of ambiguities, typos, or semantic ambiguities in user input by suggesting alternate query options. At the same time, update the LLM prompt to better select databases and output the appropriate queries/commands, while also ensuring that the LLM outputs are consistent.

Second, improve the visualization for our interaction portion. Right now, because our interaction section only stops if the user asks to exit, and if the user has asked too many questions previously, our jupyter notebook page will display a long line of questions and output results with scroll clutter, making it difficult for users to maintain track of and identify their current involvement. As a result, we planned to compress that interaction area by maintaining only 1-5 responses at a time in order to provide a better and clearer view. Furthermore, we want to implement a terminating method other than asking the user to enter exit by enabling our system to turn off automatically if the user fails to respond to our system in 10 minutes. Eventually, we want to create a front-end UI. While the current system interacts with the command line, we intend to develop a graphical user interface (GUI) and use frameworks like Streamlit or Flask to deploy a simple web application that allows users to interact with queries via a browser without displaying our coding.

Google Drive Link

https://drive.google.com/drive/folders/1GyqpTJJkOyHA4eiYM08rKmWzcb1daD3k?usp=sharing