Final Report

Jeong Hoon (Sian) Choi

December 11, 2024

Due Date:	December 13, 2024
Group:	ChatDB 94
Instructor:	*Prof.* Wensheng Wu

Name:	Jeong Hoon (SIAN) Choi (5023184813)
Phone:	+01 (323)-630-6334
Email1:	choijeon@usc.edu
Email2:	csian7386@gmail.com
Web Page:	https://csian98.github.io
Project Github:	https://github.com/csian98/StradIAN

1 Introduction

This project was conducted over the **Fall semester of DSCI 551 2024**, aiming to design a program for learning queries on database system. The system was designed to help user interact with a database without requiring deep knowledge of SQL, while also supporting natural language queries and generating sample SQL queries. The primary focus was on enabling customers and managers in an automated trading system (StradIAN) for cryptocurrencies, currency, and indices to retrieve and query data easily. StradIAN is a trading automation program that the author began developing in the Fall. ChatDB was created to enhance the program, specifically improving its insufficient user interface.

2 Planned Implementation

The project, ChatDB, provides user convenient access to a database system used by a robo-advisor. The robo-advisor system utilizes MariaDB as the DBMS, running on an Arch Linux server environment. The data to be used in this system includes:

- Market Data (stock, Cryptocurrency, Exchange Rate, etc.)
- Customer Information (Investment amount, Share, ID, email, etc.)
- System data
- Liquidity data for funds under management (market asset ratio, deposit, withdrawal, profit for each market, etc.)

The data is stored and structured in JSON files located in `etc/json/<database>/<table>.json`, where each JSON file corresponds to a table in the database.

Key tasks for the project are:

- Configuring data and execution environment (using crawlers)
- Designing the user interface
- Exploring SQL databases
- Generating sample queries with specific language constructs
- Pattern matching algorithm using JSON parsing
- Example SQL execution
- Generating natural
- Finalizing the user interface and functionalities

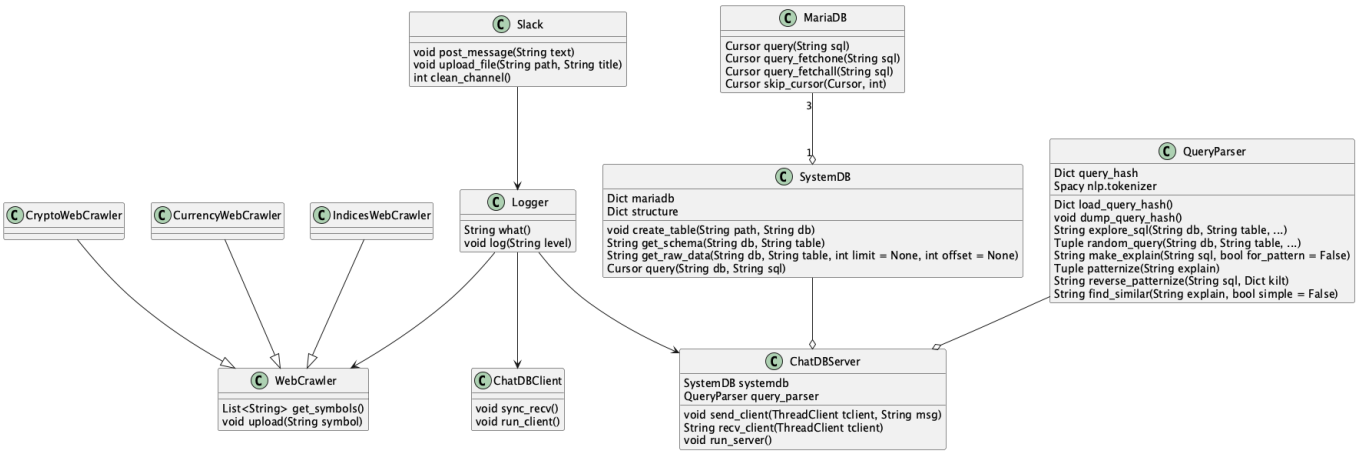
(Considering usign OpenAPI or Llama 3.2 for natural language processing)

3 Architecture Design

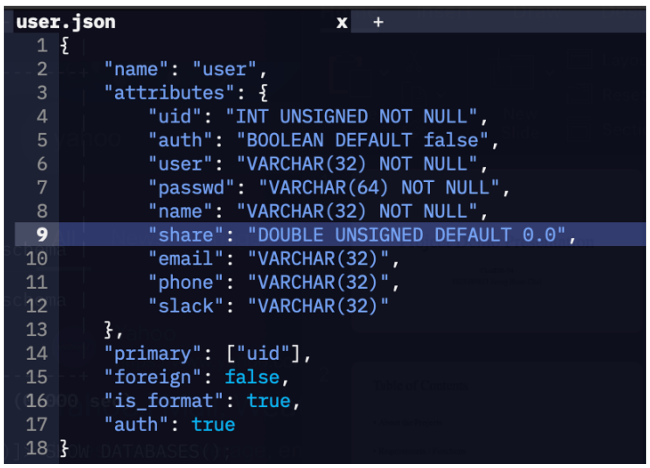
The project was devloped and test on an Arch Linux server environment in South Korea. MariaDB 11.6.2 was used as the DBMS, and Python 3.12.7 served as the development environment. Data Stored in mariaDB or retrieved through the crawler is stored as JSON files following the table schema.

The development environment details are as follows:

- CPU: AMD Ryzen9 4 Gen 5900X
- MainBoard: ASUS ROR Strix B550-XE
- RAM: G.Skill DDR4-3200 CL16 (x2)
- VGA: Nvidia ASUS ProArt RTX 4060 8 GB
- OS: 6.12.4-arch1-1



The development was done using **Python** code, and the object developed are as shown above. The core objects are **SystemDB** and **QueryParser**. **SystemDB** is a wrapper for the MariaDB class and is used to retrieve the structure of the stored database or fetch data by executing queries. **QueryParser** uses the information obtained from the tables in **SystemDB** to generate sampel queries and is also used for natural language parsing and query generation.



These JSON files are stored in the `etc/json/` directory under the respective database name and table names. Data stored in DBMS or to be stored through a crawler is stored as a JSON file with the table name in the database name folder in etc/json, and the table schema follows the JSON file.

4 Implementation

4.1 Functionalities

ChatDB was developed to meet the requirements of the project.
It includes functionalities to:

- Explore databases
- Obtain sample SQL queries
- Generate SQL queries with specific language constructs
- Convert natural language questions into SQL queries

Additionally, the system is designed as a server-client application, where different permissions can be granted based on user login, allowing users to query specific tables within their permissions. Crawlers are also used to store additional data, making the system adaptable to a variety of use cases.
(The crawler collects and stores all available data from the `system.<market type>_market` table, targeting the data where `trade = true` is specified.)

4.2 Tech Stack

The system was built using the following technologies:
Software

- Arch Linux 6.12.4-arch1-1
- Python 3.12.7
- MariaDB 11.6.2

Libraries
- beautifulsoup4 4.12.3
- mariadb 1.1.11
- nltk 3.9.1
- numpy 2.0.2
- regex 2024.11.6
- selenium 4.27.1
- spacy 3.8.3
- wget 3.2

4.3 Implementation Screenshots

Server and Client run & Login
To use the system based on the assumed scenario, the interface is designed with a server-client architecture. For the server to run, the files `etc/c2c/stradian.key` and `etc/c2c/stradian.crt` are required.
(Sample Administrator Account; ID: root | PW: root)

```
(python_stradian) [stradian@archLinux-giovanni StradIAN]$ python pylib/exec/chatdb_server_main.py &
[1] 57006
(python_stradian) [stradian@archLinux-giovanni StradIAN]$ python pylib/exec/chatdb_client_main.py

================================================

                    StradIAN
                     ChatDB

================================================
user:
```

```
================================================

                    StradIAN
                     ChatDB

================================================
user: csian7386
pswd:

Login Success

= MENU ========================================
1) explore database
2) SQL queries
8) admin
9) logout
0) exit
[$adm-Jeong Hoon Choi] ~>>
```

Explore DataBase
The **SystemDB** object has a *Dict* structure attribute. It stores all databases, tables, columns (column name, QLT || QNT,

detailed types, and permissions), and is used as the default in all operations, including Explore DataBase. The data is uploaded from a JSON file. Explore DataBase provides functionality for viewing the schema and data (with OFFSET and LIMIT options).

```
= MENU ======================================
1) explore database
2) SQL queries
8) admin
9) logout
0) exit
[$adm-Jeong Hoon Choi] ~>> 1
= EXPLORE DATABASE ==========================
1. system
2. crypto_1d
3. crypto_1h
4. currency_1d
5. indices_1d

Select the database to explore ('q' to return)
[$adm-Jeong Hoon Choi] ~>> 2
= SHOW TABLES ===============================
1. BNBUSDT
2. BTCUSDT
3. ETHUSDT
4. SOLUSDT
5. XRPUSDT

Select the table to explore ('q' to return)
[$adm-Jeong Hoon Choi] ~>>
```

```
= EXPLORE DATA ==================================
1) Schema
2) All Data <LIMIT;optional> <OFFSET;optional>
Select data to explore ('q' to return)
[$adm-Jeong Hoon Choi] ~>> 1
+-------------------+-------------------+-----+-----+---------+-------+
|              Field|               Type| Null|  Key|  Default|  Extra|
+-------------------+-------------------+-----+-----+---------+-------+
|           opentime|          timestamp|   NO|  PRI|     None|       |
|               open|     double unsigned|   NO|     |     None|       |
|               high|     double unsigned|   NO|     |     None|       |
|                low|     double unsigned|   NO|     |     None|       |
|              close|     double unsigned|   NO|     |     None|       |
|             volume|     double unsigned|   NO|     |     None|       |
|          closetime|          timestamp|   NO|     |     None|       |
|  quote_asset_volume|     double unsigned|   NO|     |     None|       |
|    number_of_trades|    int(10) unsigned|   NO|     |     None|       |
+-------------------+-------------------+-----+-----+---------+-------+
= EXPLORE DATA ==================================
1) Schema
2) All Data <LIMIT;optional> <OFFSET;optional>
Select data to explore ('q' to return)
[$adm-Jeong Hoon Choi] ~>> 2 10 100
|          opentime|              open|              high|               low|             close|            volume|          closetime|  quote_asset_volume|    number_of_trades|
| 2017-11-25 00:00:00|         8138.990000|         8734.780000|         8090.000000|         8700.010000|         4292.623682| 2017-11-25 23:59:59|     36093005.201402|               18632|
| 2017-11-26 00:00:00|         8700.040000|         9350.000000|         8604.720000|         9128.020000|         4147.380237| 2017-11-26 23:59:59|     37138533.839419|               19268|
| 2017-11-27 00:00:00|         9128.000000|         9654.280000|         9112.040000|         9650.000000|         4521.625707| 2017-11-27 23:59:59|     42961064.952029|               22806|
| 2017-11-28 00:00:00|         9650.000000|         9939.000000|         9570.500000|         9896.800000|         4917.210985| 2017-11-28 23:59:59|     48235644.057655|               18923|
| 2017-11-29 00:00:00|         9896.790000|        11300.030000|         8520.000000|         9687.880000|        13352.538715| 2017-11-29 23:59:59|    135976167.457579|               47894|
| 2017-11-30 00:00:00|         9687.880000|        10900.000000|         8850.800000|         9838.960000|         9389.574329| 2017-11-30 23:59:59|     91430904.341164|               41153|
| 2017-12-01 00:00:00|         9837.000000|        10898.000000|         9380.000000|        10782.990000|         6134.923633| 2017-12-01 23:59:59|     62260697.582916|               32375|
| 2017-12-02 00:00:00|        10775.040000|        11190.000000|        10620.000000|        10890.010000|         4765.439757| 2017-12-02 23:59:59|     52046689.840070|               29694|
| 2017-12-03 00:00:00|        10902.690000|        11825.000000|        10500.000000|        11165.410000|         5346.636524| 2017-12-03 23:59:59|     60350708.293328|               39335|
| 2017-12-04 00:00:00|        11165.410000|        11600.000000|        10802.000000|        11579.000000|         4663.424562| 2017-12-04 23:59:59|     52814985.639931|               32232|
= EXPLORE DATA ==================================
1) Schema
2) All Data <LIMIT;optional> <OFFSET;optional>
Select data to explore ('q' to return)
[$adm-Jeong Hoon Choi] ~>>
```

Obtain Sample Queries

The obtain Sample Queries feature generates sample queries divided into six keywords: *default*, *where*, *group_by*, *having*, *join*, *order_by*, and provides explanations for each. Additionally, it offers functionalities for regenerating radom samples using a keyword and executing the generated queries. The **SystemDB** object searches for available keywords for the corresponding table, while the **QueryParser** generates queries by creating random columns names, conditions, and other elements based on the available keywords. It also provides explanations for the generated queries.
(For conditional statements, when executing queries, users can directly specify numeric conditions or, if set to random, the system retrieves a random value from the table to use in the condition.)

```
= MENU ==========================================
1) explore database
2) SQL queries
8) admin
9) logout
0) exit
[$adm-Jeong Hoon Choi] ~>> 2
= QUERY DATABASE ================================
1. system
2. crypto_1d
3. crypto_1h
4. currency_1d
5. indices_1d

Select the database to query ('q' to return)
[$adm-Jeong Hoon Choi] ~>> 1
= SHOW TABLES ====================================
1. asset
2. balance
3. crypto_market
4. currency_market
5. indices_market
6. market
7. market_type
8. user

Select the table to query ('q' to return)
```

```
= SAMPLE QUERIES ================================
1) default
  SELECT * FROM `asset`;

  show the all in the `asset` table
2) where
  SELECT qty FROM `asset` WHERE qty >= <WWHERE>;

  show the qty in the `asset`  table with qty >= <WWHERE>
3) group_by
  SELECT type, MAX(qty) FROM `asset` GROUP BY type;

  show the type, max value of qty in the `asset`  table by type group
4) having
  SELECT type, AVG(qty) FROM `asset` GROUP BY type HAVING STD(qty) = <@HAVING>;

  show the type, average of qty in the `asset`  table by type  group that is STD(qty) = <AHAVING>
5) join
  SELECT l.type, l.symbol, l.qty FROM `asset` AS l LEFT JOIN `market_type` AS r ON l.type = r.type;

  show the left table's type, left table's symbol, left table's qty in the `asset`  table (called as l ) joining the `market_type` AS r  table that have l.type = r.type
6) order_by
  SELECT * FROM `asset` ORDER BY qty DESC;

  show the all in the `asset`  table in descending order of qty DESC
==================================================
1) Another queries
2) Run Query   >>> 2 <QUERY NUM> <LIMIT;optional> <OFFSET;optional>
Select command to try ('q' to return)
[$adm-Jeong Hoon Choi] ~>>
```


```
= SAMPLE QUERIES ================================
1) default
  SELECT * FROM `asset`;

  show the all in the `asset` table
2) where
  SELECT qty FROM `asset` WHERE qty >= <WWHERE>;

  show the qty in the `asset`  table with qty >= <WWHERE>
3) group_by
  SELECT type, MAX(qty) FROM `asset` GROUP BY type;

  show the type, max value of qty in the `asset`  table by type group
4) having
  SELECT type, AVG(qty) FROM `asset` GROUP BY type HAVING STD(qty) = <@HAVING>;

  show the type, average of qty in the `asset`  table by type  group that is STD(qty) = <AHAVING>
5) join
  SELECT l.type, l.symbol, l.qty FROM `asset` AS l LEFT JOIN `market_type` AS r ON l.type = r.type;

  show the left table's type, left table's symbol, left table's qty in the `asset`  table (called as l ) joining the `market_type` AS r  table that have l.type = r.type
6) order_by
  SELECT * FROM `asset` ORDER BY qty DESC;

  show the all in the `asset`  table in descending order of qty DESC
==================================================
1) Another queries
2) Run Query   >>> 2 <QUERY NUM> <LIMIT;optional> <OFFSET;optional>
Select command to try ('q' to return)
[$adm-Jeong Hoon Choi] ~>> 1
= SAMPLE QUERIES ================================
1) default
  SELECT type, symbol, qty FROM `asset`;

  show the type, symbol, qty in the `asset` table
2) where
  SELECT type FROM `asset` WHERE type < <WWHERE>;

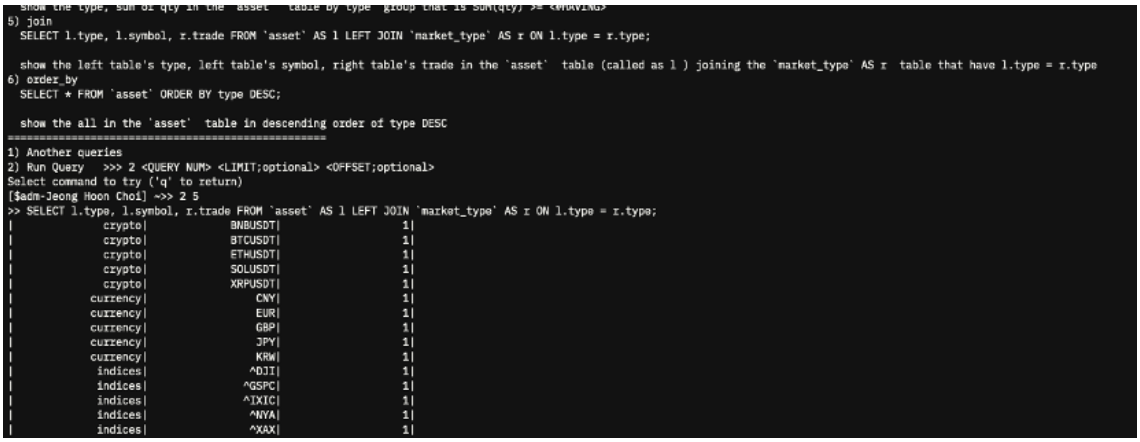
  show the type in the `asset`  table with type < <WWHERE>
3) group_by
  SELECT type, SUM(qty) FROM `asset` GROUP BY type;

  show the type, sum of qty in the `asset`  table by type group
4) having
  SELECT type, SUM(qty) FROM `asset` GROUP BY type HAVING SUM(qty) >= <HAVING>;

  show the type, sum of qty in the `asset`  table by type  group that is SUM(city) >= <P>HAVING>
5) join
  SELECT l.type, l.symbol, r.tzade FROM `asset` AS l LEFT JOIN `market_type` AS r ON l.type = r.type;

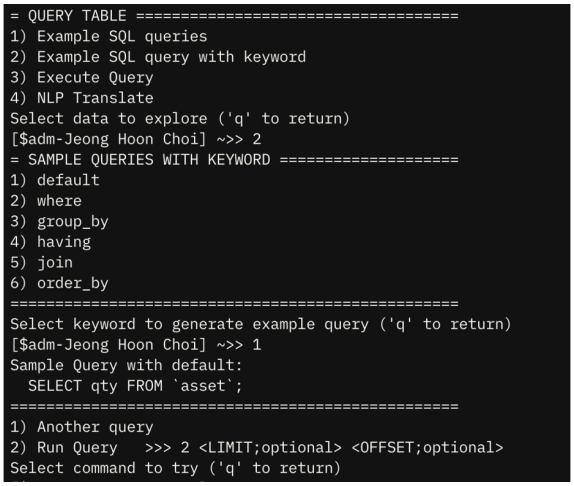
  show the left table's type, left table's symbol, right table's tzade in the `asset`  table (called as l ) joining the `market_type` AS r  table that have l.type = r.type
6) order_by
  SELECT * FROM `asset` ORDER BY type DESC;

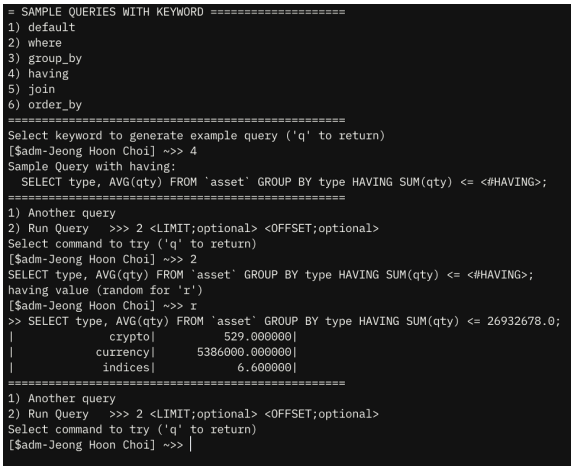
  show the all in the `asset`  table in descending order of type DESC
==================================================
1) Another queries
2) Run Query   >>> 2 <QUERY NUM> <LIMIT;optional> <OFFSET;optional>
Select command to try ('q' to return)
[$adm-Jeong Hoon Choi] ~>>
```



Obtain Sample Queries with Specific Language Constructs

The Obtain Sample Queries with Specific Language Constructs feature is identical to the Obtain Sample Queries functionality, except it allows user to select specific keywords. *(The keyword selection options are tailored to the actual available options for the given table, ensuring they are executable.)*





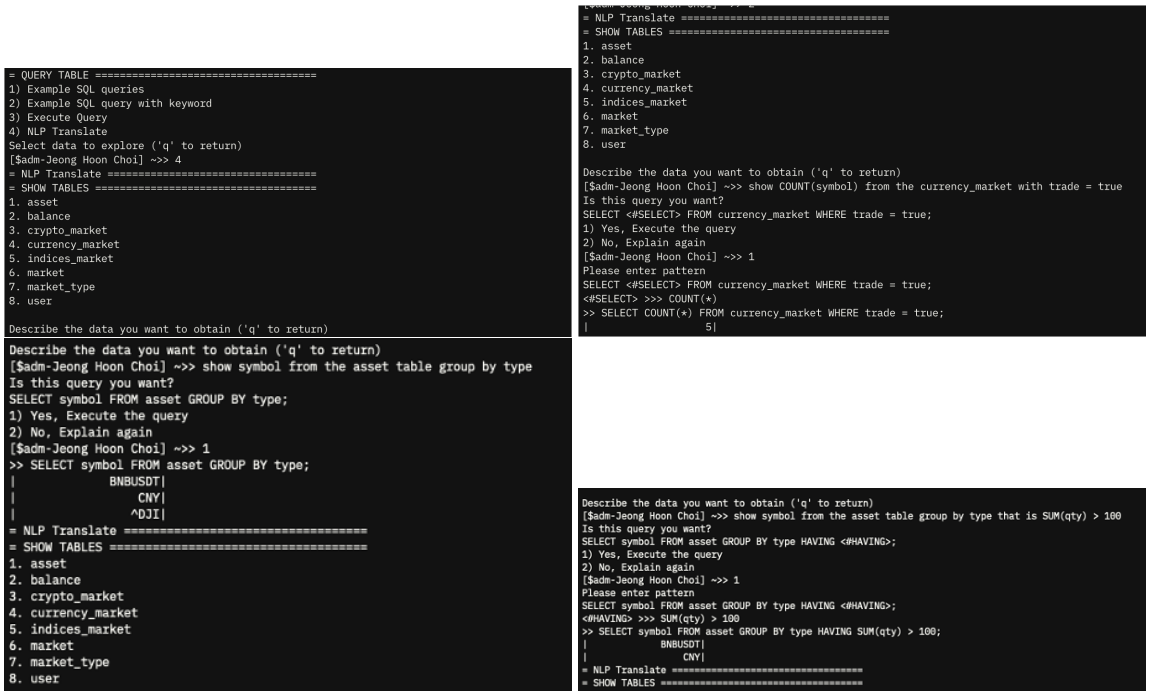
Ask Questions in Natural Language

When a user describe the desired data in natural language, the system outputs the most appropriate query. The **QueryParser** process the input natural language by parsing it into a structured format based on the **template**:

SELECT <#SELECT> FROM <#FROM> (JOIN <#JOIN> ON <#ON>) (WHERE <#WHERE>) (GROUP BY <#GROUPBY> (HAVING <#HAVING>)) (ORDER BY <#ORDERBY>)

(Parentheses indicate optional components.)

The natural languge input is parsed to create a **kilt**, a dictionary structure with key(<#*>) corresponding to their respective real values and the template. **QueryParser** matches the parsed **kilt** against pre-patterned explanations using **Jaccard similarity**. The most similar explanation is used to refine and replace the **template** structure with the appropriate **kilt** values. During the sample query generation process, patterned explanastions and extracted **templates** are stored as a dictionary for future use. *(To enhance response flexibility, additional descriptions are stored in* `etc/query/query_explain.json` *).* The natural language description must include precise table and column names to generate valid queries. If any **kilt** value cannot be replaced during parsing, the system prompts the user to manually input the missing value, allowing the query to be completely and executed.



5 Learning Outcomes

5.1 Challenges Faced

The most challenging aspect of the project was generating queries from natural language descriptions. Due to the restriction against using deep learning models like LLMs for more generalized query generation and interpreting natural languages, I had to rely on pattern matching and manually coded assumptions. This approach required crafting query templates and extracting attributes and conditions directly from user descriptions to insert them into these template. The pattern matching process involved using numerous *if-else if-else* conditional statements to anticipate various possible explanations and pre-code them. As a result, the process depended heavily on predefined patterns, leading to difficulties in handling input descriptions that did not align with these assumptions. This limitation often prevented the system from returning appropriate queries for more flexible or unstructured descriptions.

6 Conclusion

This project was developed to support the user interface in a robo-advisor system. The program successfully fulfills the primary requirements of the project, including **Explore Database**, **Obtain Sample Queries**, **Obtain Sample Queries with Specific Language Constructs**, and **Ask Questions in Natural Language**. Additionally, it provides features such as a crawler for data storage and a login-based user interface (terminal environment), enhancing the system's functionality and flexibility.

Before this project, I had never studied SQL and database systems as intensively as I did during this semester. Initially, I doubted how a language with fewer commands and limited flexibility compared to general-purpose languages likes C++ or Python, I learned the logical power of SQL, which allows for the creation and execution of infinite query variations using its set of keywords. This experience provided me with valuable insights into the structure and capabilities of database systems.

7 Future Scope

There is still room for improvement in the query generation function for natural language explanations, particularly in terms of handling more generalized queries, I plan to enhance this feature in the future. For broader explanations, leveraging large language models (LLMs) seems to be the most effective approach to accurately interpret natural language and generate the corresponding queries. Since the task at hand is less complex compared to other LLM applications, I am confident that I can achieve more flexible question interpretation and query generation by using open-source models, such as Meta Llama 3.2 1B, or by fine-tuning a model specifically tailored for this project.
The following is the result of a query made to the Meta Llama 3.2 3B model.

```
>>> Hello, you need to generate an appropriate query based on the input description.
  The structure of the database and table is as follows:
  [system.crypto_market(VARCHAR symbol, BOOLEAN trade; true mean trading)].
  Please generate only the query.
```sql
SELECT * FROM system.crypto_market WHERE trade = TRUE;
```

>>> show the all symbol that I can trade.
```sql
SELECT symbol
FROM system.crypto_market
WHERE trade = TRUE;
```
```

When tasks related to natural language interpretation are assigned to LLM models, they are more likely to generate appropriate queries with a higher probability, compared to the pattern matching methods that previously failed to generate the correct query for descriptions that could not be coded.