Wil Sheffield

13 December 2024

ChatDB: Learning to Query Database Systems Like ChatGPT

Introduction

Project Overview:
ChatDB is an interactive application that assists users in learning how to query SQL and NoSQL database systems. The system will support two databases: MySQL for SQL queries and MongoDB for NoSQL queries. ChatDB can suggest sample queries, execute the queries in the database systems, and display the query results to the user.

Project Goals:
- Develop an application that supports querying SQL and NoSQL databases.
- Enable the execution of generated queries within the database and display results to users.
- Allow users to upload dataset into the database.
- Implement functionality to generate sample queries for the selected database.

Project Requirements:
- ChatDB should support one SQL and one NoSQL database.
- Write an essay to research how ChatGPT makes the suggestions and how your implementation differs from that in ChatGPT.
- No usage of libraries or tools
- ChatDB should allow users to select which database to query and find out the table/collection and their attributes, along with some sample data **(Explore databases function)**
- ChatDB should allow users to ask for example database queries. It should return a set of queries that uses a variety of language constructs **(Obtain sample queries function)**
- Obtain sample queries with specific language constructs **(Group By function)**
- ChatDB should be tested on at least three datasets for each type of database system.
- For queries suggested by ChatDB, it should allow users to execute the queries in the database and obtain the query results
- In ChatDB, the data set should be stored in the database system

Languages: 1) Python 2) MySQL 3) MongoDB

Link to Google Drive with Code & Datasets:
https://drive.google.com/drive/folders/1Aq4GxLyp3pIM3CEqL7ILPSo9t_9undvj?usp=sharing

Database Design

I. SQL (MySQL) Design and In-Memory Representation

Design Overview:

- SQL Schema Emulation:
  - The in-memory SQL database mimics a relational database structure.
  - Each table consists of:
    - columns: A list of column names inferred from the input data.
    - rows: A list of dictionaries (or lists) representing the rows of the table.

Data Processing and Storage:

- Input Data Loading:
  - The open_file function reads CSV files using csv.DictReader, which automatically converts each row into a dictionary.
  - Preprocessing (preprocess_csv_data) ensures:
    - String representations of numbers are converted to int or float.
    - Null values are standardized.
  - The table name is derived from the filename ("books.csv" = "books").
- In-Memory Storage:
  - Data is stored in the global dictionary sql_data:

```
# Initialize in-memory SQL-like structure
def initialize_sql_data(file_name, user_name):
    data = open_file(file_name, user_name)
    table_name = file_name.split(".")[0]
    if data:
        sql_data[table_name] = {
            "columns": list(data[0].keys()),
            "rows": data
        }
        print(f"SQL table '{table_name}' loaded into memory.")
    return table_name
```

II. NoSQL (MongoDB) Design and In-Memory Representation

Design Overview:

- Document-Oriented Storage:
  - The NoSQL design mimics MongoDB's document-oriented structure.
  - Collections are stored as lists of JSON-like dictionaries.

Data Processing and Storage:

- Input Data Loading:
  - JSON files are read using json.load.
  - The convert_string_to_int function recursively processes the JSON data:
    - Converts string representations of numbers to int or float.
    - Retains the nested structure of the original JSON.
  - get_all_keys recursively fetches all keys from nested JSON objects ensured that not just the parents are recognized (children as well)
- In-Memory Storage:
  - Data is stored in the global dictionary mongo_db:

```
elif db_type == "mongodb":
    collection_name = file_name.split(".")[0]
    data = open_file(file_name, user_name)
    mongo_db[collection_name] = data
    print(f"MongoDB collection '{collection_name}' loaded into memory.")
```

  - collection_name: Derived from the filename (“phones.json” = “phones”).
  - data: A list of dictionaries, where each dictionary represents a document.

Datasets Used

Description of Datasets:

1. **Books CSV -**
   https://www.kaggle.com/datasets/saurabhbagchi/books-dataset

   The "Books Dataset" by Saurabh Bagchi is a subset of books available on Amazon. It includes various attributes for each book, such as title, author, genre, and other relevant details.

2. **Smartphone Sales CSV -**
   https://www.kaggle.com/datasets/juanmerinobermejo/smartphones-price-dataset

   This dataset provides a comprehensive collection of information on various smartphones, enabling a detailed analysis of their specifications and pricing. It encompasses a wide range of smartphones, encompassing diverse brands, models, and configurations

3. **E-Commerce Sales CSV -**
   https://www.kaggle.com/datasets/brsahan/e-commerce-dataset

   This e-commerce dataset provides comprehensive information on transactions, customer behavior, and revenue trends. It is ideal for exploratory data analysis (EDA) to gain insights into sales performance, customer demographics, and purchasing patterns.

4. **Phones 2024 JSON -**
   https://www.kaggle.com/datasets/jakubkhalponiak/phones-2024

   This file contains the raw data scraped from GSMArena.com, including information on various phones currently available on the market.

5. **Lottery Expenditures - Multi-Year Report JSON -**
   https://www.kaggle.com/datasets/mahdiehhajian/lottery-expenditures-multi-year-report

   This report includes expenditure data from the Oregon Lottery. The Lottery uses the Microsoft Dynamics 2009 system, a standalone ERP (enterprise resource planning system) for financial transactions.

6. **SpongeBob SquarePants Characters JSON -**
   https://www.kaggle.com/datasets/myticalcat/spongebob-squarepants-character-dataset

   This dataset contains detailed information about characters from the SpongeBob SquarePants universe. The data includes various attributes such as character appearances, occupations, relationships, and more.

Implementation Details

I. Technologies and Tools

1. **Programming Language**:
    - Python: Used for implementing the logic for query generation, database interaction, and user interface via CLI.

2. **Backend Functionalities**:
    - Simulated in-memory databases using Python dictionaries for SQL and MongoDB abstractions.
    - File handling capabilities to process .csv (for relational data) and .json (for document-based data) datasets.
    - Data preprocessing and type inference functionalities for dynamic query generation.

3. **Database Systems**:
    - SQL-like structure to mimic relational databases (tables, columns, rows).
    - MongoDB-like structure to represent NoSQL document collections.

4. **Frontend Interface**:
    - Command-Line Interface (CLI) for interaction, allowing users to upload datasets, explore database schemas, and generate sample queries.

II. Flow of Interaction

1. **User Interaction**:
    - The user selects the database type (SQL or MongoDB) and uploads the dataset in .csv or .json format.
    - The system dynamically loads and preprocesses the dataset into an in-memory structure.

2. **Data Preprocessing**:
    - For SQL-like databases:
        - Data is structured into tables with columns and rows.
        - Data types are inferred, and numeric columns are identified for advanced queries.
    - For MongoDB-like databases:
        - Data is organized into collections with nested document structures.
        - Keys and data types are extracted for schema identification.

3. **Query Generation**:
    - SQL:
        - Generates templates for constructs like GROUP BY, HAVING, WHERE, and ORDER BY.
        - Ensures queries align with the dataset's structure and inferred data types.
    - MongoDB:
        - Creates queries using constructs like find, $group, $match, and $sort.
        - Utilizes field-type inference for conditionals and aggregations.

4. **Exploration and Query Execution**:
    - Users can explore the schema and preview data.
    - Query results are simulated and displayed in a structured format for the user to review.

5. **Dynamic Features**:
    - Randomized query templates ensure variability.
    - Natural language descriptions accompany queries to enhance understanding.

III. Example Workflow

1. The user launches the application and selects SQL or MongoDB.
2. A dataset is uploaded and processed into an appropriate in-memory structure.
3. The user explores the database to view its schema and sample data.
4. Query templates are generated dynamically, and users can execute these queries to simulate results.
5. The interaction continues until the user opts to exit the program.

Systematic Strategy for Generating Query Templates

The query generation strategy in ChatDB systematically creates dynamic templates based on the structure of the dataset and database type (SQL or NoSQL). The templates are designed to showcase the variety of operations users can perform, such as filtering, grouping, sorting, and aggregation.

I. Key Components of the Strategy

1. **Data Exploration**:
    - Extract schema details (e.g., columns for SQL tables or keys for NoSQL documents).
    - Identify numeric and non-numeric fields for appropriate query operations.

2. **Dynamic Template Generation**:
    - Use randomization to select columns or keys for query generation to ensure variability.
    - Adapt query templates to the dataset's structure (e.g., avoid operations on non-numeric columns for numeric aggregations).

3. **Natural Language Descriptions**:
    - Pair each query template with a natural language explanation to make the query purpose clear.

4. **Output**:
    - Generate query results using Python's data manipulation capabilities to help users visualize expected outcomes

II. SQL Query Templates

For SQL, the following types of queries are systematically generated:

1. **GROUP BY Template**:
    - Query & Natural Language In Code:

```
query = f"SELECT {group_column}, COUNT(*) FROM {table_name} GROUP BY {group_column};"
nl = f"Count the number of rows grouped by {group_column}."
```

2. **HAVING Template:**
    - Query & Natural Language In Code:

```
query = f"SELECT {group_column}, AVG({numeric_col}) FROM {table_name} GROUP BY {group_column} HAVING AVG({numeric_col}) > {random_threshold}
nl = f"Find rows where the average of {numeric_col} is greater than {random_threshold}, grouped by {group_column}."
```

3. **ORDER BY Template:**
    - Query & Natural Language In Code:

```
query = f"SELECT {order_column} FROM {table_name} ORDER BY {order_column} DESC;"
nl = f"List all values of {order_column} in descending order."
```

4. **WHERE Template:**
    - Query & Natural Language In Code:

```
query = f"SELECT {output_column} FROM {table_name} WHERE {filter_column} = '{selected_value}';"
nl = f"Find rows where {filter_column} equals '{selected_value}' and display {output_column}."
```

5. **LIMIT Template:**
    - Query & Natural Language In Code:

```
query = f"SELECT {', '.join(selected_columns)} FROM {table_name} LIMIT {dynamic_limit};"
nl = f"Display the first {dynamic_limit} rows with columns {', '.join(selected_columns)}."
```

6. **JOIN Template:**
    - Query & Natural Language In Code:

```
query = (
    f"SELECT {table_name}.{selected_columns[0]}, {another_table_name}.{selected_columns[1]} "
    f"FROM {table_name} JOIN {another_table_name} "
    f"ON {table_name}.{join_column} = {another_table_name}.{join_column};"
)
nl = (
    f"Join the {table_name} table with an alias of itself ({another_table_name}) on the column {join_column}, "
    f"and display {selected_columns[0]} from the original table and {selected_columns[1]} from the alias table."
)
```

7. **LIKE Template:**
    - Query & Natural Language In Code:

```
query = f"SELECT {selected_column}, {random.choice(columns)} FROM {table_name} WHERE {selected_column} LIKE '%{substring}%';"
nl = f"Find rows where {selected_column} contains the text '{substring}' and display {selected_column} and another column."
```

8. **RANGE Template:**
    - Query & Natural Language In Code:

```
query = f"SELECT {numeric_col}, {range_column} FROM {table_name} WHERE {numeric_col} BETWEEN {lower_bound} AND {upper_bound};"
nl = f"Find rows where {numeric_col} is between {lower_bound} and {upper_bound} and display {range_column}."
```

9. **SUM Template:**
    - Query & Natural Language In Code:

```
# Generate the query
query = f"SELECT {group_column}, SUM({numeric_col}) FROM {table_name} GROUP BY {group_column};"
nl = f"Calculate the total sum of {numeric_col}, grouped by {group_column}."
```

III. MongoDB Query Templates

For MongoDB, query templates leverage constructs such as find, $group, $match, and $sort:

1. **FIND Template**:
    - Query & Natural Language In Code:

```
query = f"db.{collection_name}.find({{}})"
nl = f"Find all documents in the {collection_name} collection."
```

2. **PROJECTION Template:**
    - Query & Natural Language In Code:

```
query = f"db.{collection_name}.find({{}}, {{ {', '.join([f'{field}: 1' for field in projection_fields])}, _id: 0 }})"
nl = f"Find all documents and display only {', '.join(projection_fields)}."
```

3. **CRITERIA Template:**
    - Query & Natural Language In Code:

```
query = f"db.{collection_name}.find({{ {numeric_field}: {{ $gt: {random_threshold} }} }})"
nl = f"Find documents where {numeric_field} is greater than {random_threshold}."
```

4. **CONDITIONS Template:**
    - Query & Natural Language In Code:

```
query = f"db.{collection_name}.find({{ {numeric_field}: {{ $gt: {random_threshold} }}, {non_numeric_field}: '{selected_value}' }})"
nl = f"Find documents where {numeric_field} is greater than {random_threshold} and {non_numeric_field} equals '{selected_value}'."
```

5. **MATCH Template:**
    - Query & Natural Language In Code:

```
query = f"db.{collection_name}.aggregate([{{ $match: {{ {numeric_field}: {{ $gte: {lower_bound}, $lte: {upper_bound} }} }} }}])"
nl = f"Find documents where {numeric_field} is between {lower_bound} and {upper_bound}."
```

6. **GROUP/SUM Template:**
    - Query & Natural Language In Code:

```
query = (
    f"db.{collection_name}.aggregate(["
    f"{{ $group: {{ _id: '${group_field}', total: {{ $sum: '${sum_field}' }} }} }}])"
)
nl = f"Group documents by {group_field} and calculate the sum of {sum_field}."
```

7. **SORT/LIMIT Template:**
    - Query & Natural Language In Code:

```
query = (
    f"db.{collection_name}.aggregate(["
    f"{{ $sort: {{ {sort_field}: 1 }} }}, "  # Sort in ascending order
    f"{{ $limit: {dynamic_limit} }}])"
)
nl = f"Sort documents by {sort_field} in ascending order and return the top {dynamic_limit}."
```

Query Execution Strategy

I. SQL Query Execution Strategy

- **1) GROUP BY Execution**
  - Purpose: Count the occurrences of unique values in a specific column.
  - Execution Steps:
    - 1) A column is selected for grouping.
    - 2) A Counter is used to count the occurrences of each unique value in the selected column.
    - 3) The result is returned as a dictionary of {group_value: count}.

```
group_counts = Counter(row[group_column] for row in rows)
simulated_output = dict(group_counts)
```

  - Sample Execution:

```
Count the number of rows grouped by Storage.
SQL Query: SELECT Storage, COUNT(*) FROM smartphones_sales GROUP BY Storage;
Would you like to execute this query? (yes/no): yes
Query Output:
{2: 2,
 3: 2,
 4: 6,
 8: 3,
 12: 1,
 16: 24,
 32: 166,
 64: 332,
 128: 732,
 256: 412,
 512: 94,
 1000: 17,
 '': 25}
```

- **2) HAVING Execution**
  - Purpose: Filter groups based on an aggregate condition (e.g., average greater than a threshold).
  - Execution Steps:
    - 1) For each unique value in the group column, calculate the aggregate (e.g., average) for the numeric column.
    - 2) Include groups that meet the threshold condition in the result.

```
simulated_output = {}
for group in set(row[group_column] for row in rows if row[group_column] is not None):
    group_rows = [row for row in rows if row[group_column] == group]
    group_avg = sum(float(row[numeric_col]) for row in group_rows if row[numeric_col] is not None) / len(group_rows)
    if group_avg > random_threshold:
        simulated_output[group] = group_avg
```

  - Sample Execution:

```
Find rows where the average of Total Price is greater than 3323.93, grouped by Product Name.
SQL Query: SELECT Product Name, AVG(Total Price) FROM e-commerce_sales GROUP BY Product Name HAVING AVG(Total Price) > 3323.93;
Would you like to execute this query? (yes/no): yes
Query Output:
{'Laptop': 4519.480519480519}
```

- **3) ORDER BY Execution**
  - Purpose: Sort rows based on a specified column in ascending or descending order.
  - Execution Steps:
    - 1) Extract values from the selected column.
    - 2) Sort the values in descending order.

```
# Filter and normalize values for sorting
valid_values = [row[order_column] for row in rows if isinstance(row[order_column], (int, float, str))]
normalized_values = [str(value) for value in valid_values]  # Convert all values to strings for sorting
simulated_output = sorted(normalized_values, reverse=True)
```

  - Sample Execution:

```
List all values of Customer ID in descending order.
SQL Query: SELECT Customer ID FROM e-commerce_sales ORDER BY Customer ID DESC;
Would you like to execute this query? (yes/no): yes
Query Output:
['CUST0300',
 'CUST0300',
 'CUST0299',
 'CUST0299',
 'CUST0299',
 'CUST0299',
 'CUST0299',
 'CUST0299',
 'CUST0299',
 'CUST0298',
 'CUST0298',
 'CUST0298',
 'CUST0298',
 'CUST0297',
 'CUST0297',
```

- **4) WHERE Execution**
  - Purpose: Filter rows based on a condition for a specific column.
  - Execution Steps:
    - 1) Iterate through rows and check if the value in the condition column matches the specified value.
    - 2) Include rows meeting the condition in the output.

```
simulated_output = [row[output_column] for row in rows if row[filter_column] == selected_value]
```

  - Sample Execution:

```
Find rows where Age equals '50.0' and display Total Price.
SQL Query: SELECT Total Price FROM e-commerce_sales WHERE Age = '50.0';
Would you like to execute this query? (yes/no): yes
Query Output:
[100.0,
 250.0,
 1000.0,
 500.0,
 7500.0,
 800.0,
 300.0,
 4000.0,
 900.0,
 1200.0,
 30.0,
 2400.0,
 900.0,
 200.0,
 300.0,
 1500.0,
 120.0,
 150.0,
 300.0,
 200.0,
 800.0,
 90.0,
 1200.0]
```

- **5) LIMIT Execution**
  - Purpose: Return a limited number of rows from the dataset.
  - Execution Steps:
    - 1) Slice the rows list up to the specified limit.
    - 2) Include selected columns in the output.

```
simulated_output = [
    {col: row[col] for col in selected_columns} for row in rows[:dynamic_limit]
]
```

  - Sample Execution:

```
Display the first 7 rows with columns Gender, Product Name.
SQL Query: SELECT Gender, Product Name FROM e-commerce_sales LIMIT 7;
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Gender': 'Male', 'Product Name': 'Monitor'},
 {'Gender': 'Male', 'Product Name': 'Headphones'},
 {'Gender': 'Female', 'Product Name': 'Monitor'},
 {'Gender': 'Male', 'Product Name': 'Headphones'},
 {'Gender': 'Female', 'Product Name': 'Laptop'},
 {'Gender': 'Male', 'Product Name': 'Smartwatch'},
 {'Gender': 'Male', 'Product Name': 'Smartwatch'}]
```

- **6) JOIN Execution**
  - Purpose: Simulate joining two tables on a common column (currently virtual).
  - Execution Steps:
    - This is simulated and does not actually join rows due to the lack of a second table. Instead, a placeholder message is returned.

```
simulated_output = "Simulated output not available for JOIN queries because the alias table is virtual."
```

  - Sample Execution:

```
Join the e-commerce_sales table with an alias of itself (e-commerce_sales_alias) on the column Region, and display Product Name from the original table and Age from the alias table.
SQL Query: SELECT e-commerce_sales.Product Name, e-commerce_sales_alias.Age FROM e-commerce_sales JOIN e-commerce_sales_alias ON e-commerce_sales.Region = e-commerce_sales_alias.Region
;
Would you like to execute this query? (yes/no): yes
Query Output:
('Simulated output not available for JOIN queries because the alias table is '
 'virtual.')
```

- **7) LIKE Execution**
  - Purpose: Filter rows based on whether a text column contains a specific substring.
  - Execution Steps:
    - 1) Select a text-based column and generate a substring from a sample value.
    - 2) Iterate through rows and include those where the column contains the substring.
    - 3) Include additional selected columns in the result.

```
simulated_output = [
    {selected_column: row[selected_column], columns[1]: row[columns[1]]}
    for row in rows if substring.lower() in str(row[selected_column]).lower()
]
```

  - Sample Execution:

```
Find rows where Gender contains the text 'Mal' and display Gender and another column.
SQL Query: SELECT Gender, Quantity FROM e-commerce_sales WHERE Gender LIKE '%Mal%';
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Gender': 'Male'},
 {'Gender': 'Male'},
 {'Gender': 'Female'},
 {'Gender': 'Male'},
 {'Gender': 'Female'},
 {'Gender': 'Male'},
 {'Gender': 'Male'},
 {'Gender': 'Female'},
 {'Gender': 'Female'},
 {'Gender': 'Male'},
 {'Gender': 'Female'},
 {'Gender': 'Female'},
 {'Gender': 'Female'},
 {'Gender': 'Male'},
 {'Gender': 'Male'},
 {'Gender': 'Female'},
```

- **8) RANGE Execution**
  - Purpose: Filter rows based on whether a numeric column's value falls within a specified range.
  - Execution Steps:
    - 1) Select a numeric column and calculate a range (lower and upper bounds).
    - 2) Iterate through rows, including those whose numeric column values fall within the range.
    - 3) Include additional selected columns in the result.

```
simulated_output = [
    {numeric_col: row[numeric_col], range_column: row[range_column]}
    for row in rows if lower_bound <= float(row[numeric_col]) <= upper_bound
]
```

  - Sample Execution:

```
Find rows where Quantity is between 1.29 and 3.46 and display Unit Price.
SQL Query: SELECT Quantity, Unit Price FROM e-commerce_sales WHERE Quantity BETWEEN 1.29 AND 3.46;
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Quantity': 2.0, 'Unit Price': 100.0},
 {'Quantity': 3.0, 'Unit Price': 1500.0},
 {'Quantity': 2.0, 'Unit Price': 200.0},
 {'Quantity': 2.0, 'Unit Price': 50.0},
 {'Quantity': 3.0, 'Unit Price': 100.0},
 {'Quantity': 2.0, 'Unit Price': 100.0},
 {'Quantity': 3.0, 'Unit Price': 30.0},
 {'Quantity': 3.0, 'Unit Price': 200.0},
 {'Quantity': 2.0, 'Unit Price': 800.0},
 {'Quantity': 3.0, 'Unit Price': 30.0},
 {'Quantity': 3.0, 'Unit Price': 30.0},
 {'Quantity': 2.0, 'Unit Price': 100.0},
 {'Quantity': 2.0, 'Unit Price': 1658.4323266595475},
 {'Quantity': 2.0, 'Unit Price': 300.0},
 {'Quantity': 2.0, 'Unit Price': 300.0},
 {'Quantity': 3.0, 'Unit Price': 30.0},
 {'Quantity': 2.0, 'Unit Price': 800.0},
 {'Quantity': 2.0, 'Unit Price': 300.0},
 {'Quantity': 2.0, 'Unit Price': 800.0},
 {'Quantity': 2.0, 'Unit Price': 50.0},
 {'Quantity': 3.0, 'Unit Price': 50.0},
 {'Quantity': 2.0, 'Unit Price': 100.0},
 {'Quantity': 2.0, 'Unit Price': 300.0},
```

- **9) SUM Execution**
  - Purpose: Group rows by a specific column and calculate the sum of a numeric column within each group.
  - Execution Steps:
    - 1) Group rows by a selected column.
    - 2) For each group, calculate the sum of the numeric column.
    - 3) Return a dictionary of {group_value: total_sum}.

```
simulated_output = {
    group: sum(float(row[numeric_col]) for row in rows if row[group_column] == group and row[numeric_col] is not None)
    for group in set(row[group_column] for row in rows if row[group_column] is not None)
}
```

  - Sample Execution:

```
Calculate the total sum of Unit Price, grouped by Gender.
SQL Query: SELECT Gender, SUM(Unit Price) FROM e-commerce_sales GROUP BY Gender;
Would you like to execute this query? (yes/no): yes
Query Output:
{'Female': 218835.14860903518, 'Male': 238868.628404873}
```

II. MongoDB Query Execution Strategy

- **1) FIND Execution:**
  - Purpose: Retrieve all documents from a collection.
  - Execution Steps:
    - 1) Fetch the collection specified by collection_name.
    - 2) Return the first n documents, where n is set to default of 5.

```
simulated_output = mongo_db[collection_name][:5]
```

  - Sample Execution:

```
Find all documents in the lottery_expenditures collection.
MongoDB Query: db.lottery_expenditures.find({})
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Acct Name': 'Utilities',
  'Amount': 2399.4,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2021,
  'GL Acct': 79020,
  'State': 'OR',
  'Vendor Name': 'SANIPAC INC'},
 {'Acct Name': 'Rent',
  'Amount': 17215.68,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2019,
  'GL Acct': 'OR',
  'State': ' LLC"',
  'Vendor Name': '"BY FAMILY'},
 {'Acct Name': 'Agency Fees',
  'Amount': 10500,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2021,
  'GL Acct': 85010,
  'State': 'OR',
  'Vendor Name': 'BRIGHTWATER ENTERTAINMENT INC'},
```

- **2) PROJECTION Execution:**
  - Purpose: Select specific fields from documents in the collection.
  - Execution Steps:
    - 1) Dynamically select up to 2 fields for projection from the document keys.
    - 2) For each document, extract only the selected fields.
    - 3) Return the modified documents with only the projected fields.

```
simulated_output = [{key: doc.get(key) for key in projection_fields} for doc in mongo_db[collection_name][:5]]
```

  - Sample Execution:

```
Find all documents and display only Acct Name, Amount.
MongoDB Query: db.lottery_expenditures.find({}, { Acct Name: 1, Amount: 1, _id: 0 })
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Acct Name': 'Utilities', 'Amount': 2399.4},
 {'Acct Name': 'Rent', 'Amount': 17215.68},
 {'Acct Name': 'Agency Fees', 'Amount': 10500},
 {'Acct Name': 'Agency Fees', 'Amount': 5625},
 {'Acct Name': 'Agency Fees', 'Amount': 6500}]
```

- **3) CRITERIA Execution:**
  - Purpose: Filter documents based on a condition for a numeric field.
  - Execution Steps:
    - 1) Identify numeric fields in the collection.
    - 2) Dynamically select a numeric field and calculate a threshold.
    - 3) Iterate through the documents, including only those where the field value exceeds the threshold.

```
simulated_output = [
    doc for doc in mongo_db[collection_name]
    if isinstance(doc.get(numeric_field), (int, float)) and doc.get(numeric_field) > random_threshold
]
```

  - Sample Execution:

```
Find documents where Fiscal Year is greater than 2021.
MongoDB Query: db.lottery_expenditures.find({ Fiscal Year: { $gt: 2021 } })
Would you like to execute this query? (yes/no):
```

```
{'Acct Name': 'Out-of-State Travel',
 'Amount': 224,
 'Department': '177-OREGON STATE LOTTERY',
 'Fiscal Year': 2024,
 'GL Acct': 71030,
 'State': 'Oregon',
 'Vendor Name': 'TINA ERICKSON'},
{'Acct Name': 'In-State Travel',
 'Amount': 84,
 'Department': '177-OREGON STATE LOTTERY',
 'Fiscal Year': 2024,
 'GL Acct': 71010,
 'State': 'Oregon',
 'Vendor Name': 'TITUS OVERTURF'},
{'Acct Name': 'Out-of-State Travel',
 'Amount': 372.32,
 'Department': '177-OREGON STATE LOTTERY',
 'Fiscal Year': 2024,
 'GL Acct': 71030,
 'State': 'Oregon',
 'Vendor Name': 'TITUS OVERTURF'},
```

- **4) CONDITIONS Execution:**
  - Purpose: Filter documents based on combined conditions for numeric and non-numeric fields.
  - Execution Steps:
    - 1) Dynamically select one numeric and one non-numeric field.
    - 2) Generate a threshold for the numeric field and a random value for the non-numeric field.
    - 3) Filter documents that satisfy both conditions.

```
simulated_output = [
    doc for doc in mongo_db[collection_name]
    if isinstance(doc.get(numeric_field), (int, float)) and doc.get(numeric_field) > random_threshold
    and doc.get(non_numeric_field) == selected_value
]
```

  - Sample Execution:

```
Find documents where Fiscal Year is greater than 2023 and Acct Name equals 'Safety'.
MongoDB Query: db.lottery_expenditures.find({ Fiscal Year: { $gt: 2023 }, Acct Name: 'Safety' })
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Acct Name': 'Safety',
  'Amount': 599,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2024,
  'GL Acct': 83055,
  'State': 'Oregon',
  'Vendor Name': 'PRO DRIVE INC'},
 {'Acct Name': 'Safety',
  'Amount': 550,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2024,
  'GL Acct': 83055,
  'State': 'Oregon',
  'Vendor Name': 'CANOPY WELLBEING C/O CASCADE CENTERS'},
 {'Acct Name': 'Safety',
  'Amount': 2204.99,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2024,
  'GL Acct': 83055,
  'State': 'Oregon',
  'Vendor Name': 'MOBILE TECH FITNESS REPAIR LLC'},
 {'Acct Name': 'Safety',
  'Amount': 414.99,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2024,
  'GL Acct': 83055,
  'State': 'North Dakota',
  'Vendor Name': 'US BANK NATIONAL ASSOCIATION ND'}]
```

- **5) MATCH Execution:**
  - Purpose: Filter documents based on whether a numeric field's value falls within a range.
  - Execution Steps:
    - 1) Dynamically select a numeric field and calculate a lower and upper bound.
    - 2) Include only documents where the field value lies within the specified range.

```
simulated_output = [
    doc for doc in mongo_db[collection_name]
    if isinstance(doc.get(numeric_field), (int, float)) and lower_bound <= doc.get(numeric_field) <= upper_bound
]
```

  - Sample Execution:

```
Find documents where Amount is between 15164256 and 23789414.
MongoDB Query: db.lottery_expenditures.aggregate([{ $match: { Amount: { $gte: 15164256, $lte: 23789414 } } }])
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Acct Name': 'Machinery and Equipment',
  'Amount': 17271450,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2022,
  'GL Acct': 19025,
  'State': 'NB',
  'Vendor Name': 'IGT CANADA SOLUTIONS ULC'},
 {'Acct Name': 'PO Clearing Account',
  'Amount': 21137683,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2024,
  'GL Acct': 24020,
  'State': 'Nevada',
  'Vendor Name': 'LNW GAMING INC'}]
```

- **6) GROUP/SUM Execution:**
  - Purpose: Group documents by a field and calculate the sum of another numeric field within each group.
  - Execution Steps:
    - 1) Dynamically select a grouping field (non-numeric) and a numeric field for aggregation.
    - 2) Iterate through the documents, grouping by the selected field.
    - 3) Calculate the sum of the numeric field for each group.

```
grouped_data = defaultdict(int)
for doc in mongo_db[collection_name]:
    group_key = doc.get(group_field, "Unknown")
    numeric_value = doc.get(sum_field, 0)
    grouped_data[group_key] += numeric_value

simulated_output = [{"_id": k, "total": v} for k, v in grouped_data.items()]
```

  - Sample Execution:

```
Group documents by Acct Name and calculate the sum of GL Acct.
MongoDB Query: db.lottery_expenditures.aggregate([{ $group: { _id: '$Acct Name', total: { $sum: '$GL Acct' } } }])
Would you like to execute this query? (yes/no): yes
Query Output:
[{'_id': 'Utilities', 'total': 6400620},
 {'_id': 'Rent', 'total': 3555450},
 {'_id': 'Agency Fees', 'total': 1612190},
 {'_id': 'Attorney General Charges', 'total': 462060},
 {'_id': 'Building and Structure', 'total': 38120},
 {'_id': 'Building Improvements and Grounds', 'total': 343170},
 {'_id': 'Building Maintenance - General', 'total': 2960370},
 {'_id': 'Building Maintenance - Preventative', 'total': 240090},
 {'_id': 'Capital Interest Expense', 'total': 182040},
 {'_id': 'Cell Phones', 'total': 2072560},
 {'_id': 'Central Gaming System', 'total': 2310700},
 {'_id': 'Computer Hardware', 'total': 266280},
 {'_id': 'Computer Software - Non-Gaming', 'total': 76200},
 {'_id': 'Computer Software - Traditional', 'total': 76160},
 {'_id': 'Contract Payments', 'total': 11475980},
 {'_id': 'Contracts Payable - Short-term', 'total': 196000},
 {'_id': 'Cost of Tickets', 'total': 258030},
 {'_id': 'Creative Production', 'total': 842350},
 {'_id': 'DAS Charges', 'total': 2250300},
 {'_id': 'Digital Media', 'total': 338290},
 {'_id': 'Digital Production', 'total': 336420},
 {'_id': 'Dues', 'total': 2496320},
 {'_id': 'Employment Hearings', 'total': 154080},
 {'_id': 'Equipment Maintenance', 'total': 1152800},
```

- **7) SORT/LIMIT Execution:**
  - Purpose: Sort documents by a field and limit the number of results.
  - Execution Steps:
    - 1) Dynamically select a field for sorting.
    - 2) Sort the documents in ascending order based on the selected field.
    - 3) Slice the sorted list to return the top n results.

```
simulated_output = sorted(
    mongo_db[collection_name],
    key=lambda doc: str(doc.get(sort_field, "")),
)[:dynamic_limit]
```

  - Sample Execution:

```
Sort documents by State in ascending order and return the top 7.
MongoDB Query: db.lottery_expenditures.aggregate([{ $sort: { State: 1 } }, { $limit: 7 }])
Would you like to execute this query? (yes/no): yes
Query Output:
[{'Fiscal Year': ''},
 {'Acct Name': 'Minor Spare Parts',
  'Amount': 50667.2,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2021,
  'GL Acct': 'NV',
  'State': '  INC."',
  'Vendor Name': '"WELLS-GARDNER TECHNOLOGIES'},
 {'Acct Name': 'Minor Spare Parts',
  'Amount': 29438.07,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2020,
  'GL Acct': 'NV',
  'State': '  INC."',
  'Vendor Name': '"WELLS-GARDNER TECHNOLOGIES'},
 {'Acct Name': 'Minor Spare Parts',
  'Amount': 33395.18,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2019,
  'GL Acct': 'NV',
  'State': '  INC."',
  'Vendor Name': '"WELLS-GARDNER TECHNOLOGIES'},
 {'Acct Name': 'PO Clearing Account',
  'Amount': 11398,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2022,
  'GL Acct': 'NV',
  'State': '  INC."',
  'Vendor Name': '"WELLS-GARDNER TECHNOLOGIES'},
 {'Acct Name': 'Minor Spare Parts',
  'Amount': 39143,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2022,
  'GL Acct': 'NV',
  'State': '  INC."',
  'Vendor Name': '"WELLS-GARDNER TECHNOLOGIES'},
 {'Acct Name': 'PO Clearing Account',
  'Amount': 142273.73,
  'Department': '177-OREGON STATE LOTTERY',
  'Fiscal Year': 2023,
  'GL Acct': 'NV',
  'State': '  INC."',
  'Vendor Name': '"WELLS-GARDNER TECHNOLOGIES'}]
```