EvaDB Project 1 Report: SQL Query Explainer

Krish Nathan

Github Repo

1 Implementation Details

The goal of this project is to create a function in EvaDB which allows the user to pass in a SQL query and receive an explanation in English of what the query does. The primary goal is to support an `EXPLAIN_SQL(query)` function within EvaDB, which is what I focused on for this phase of the project. A future extension could be to support an `EXPLAIN_QUERYPLAN(query)` function which gives an English explanation of the generated query plan.

1.1 Query Explainer using ChatGPT

The first approach I tried was to utilize a large language model (LLM) such as ChatGPT to explain what a given query does. The following snippet of Python code is what makes a request to ChatGPT to explain the query.

```
import evadb
cursor = evadb.connect().cursor()

# Run ChatGPT over the Text Summary extracted by Whisper
chatgpt_udf = """
    SELECT ChatGPT('Please explain this query', "SELECT * FROM students");
"""
response = cursor.query(chatgpt_udf).df()
```

The output of this snippet is a Pandas DataFrame containing the response, which is usually similar to "The query SELECT * FROM students is used to select every tuple in the student table". This API call takes quite a while, typically between 20 and 30 seconds.

1.2 Query Explainer using GPT4All

The second approach I tried was to utilize a local LLM using the python library GPT4All. The long request time involved with calling ChatGPT would make it infeasible for use integrated within EvaDB. Using a local LLM instead could speed up the parser to improve user experience. I used the Orca Mini 3b model, which is known to perform well on limited hardware resources. The following Python snippet shows how to use GPT4All to call the local LLM.

```
from gpt4all import GPT4All
model = GPT4All("orca-mini-3b.ggmlv3.q4_0.bin")
def explain_sql(query_string: str, response_length: int) -> str:
    f_str = "Explain what the SQL query: '{}' does".format(query_string)
    tokens = list(model.generate(f_str, max_tokens=response_length, streaming=True))
    text = "".join(tokens)
    return text

print(explain_sql("SELECT * FROM students", 50))
```

This function call asks the local LLM to explain what `SELECT * FROM students` does in 50 characters or less. Running the `expain_sql` function takes between 1 and 2 seconds, but it does not always produce the expected output. The expected output is a response similar to "The given SQL query is used to retrieve all columns from a table named students...". However, the function call occasionally returns an empty string. I believe this happens since the local LLM is much less powerful than OpenAI's instance of ChatGPT, so on some runs it's not able to correctly generate the expected response.

1.3 Query Explainer using Parser

The third apporach I tried was to write my own SQL parser which explains what the query does in English. I realize that this might be quite challenging for complex SQL queries, so I limited the scope. I decided to focus on simple SQL statements including the `SELECT, INSERT, WHERE, GROUP, ORDER` clauses. I first tokenized the input SQL query and switched on the first keyword as it specifies the type of operation `(SELECT, INSERT)`. Then I focused on parsing the columns and tables lists which are passed into `SELECT`. I also accounted for the `AS` clauses which rename the columns in the output relation. Parsing `INSERT` required isolating the columns list and the values list, which are separated by the `VALUES` keyword. I realize that column names do not need to be specified if all columns are present in the values list; this is an edge case I didn't handle. Finally, the code to handle `WHERE, ORDER, GROUP` had the same basic structure. They each required isolating the arguments of the clause, splitting the arguments into a list, and formatting the arguments appropriately. For more details, look in this Jupyter Notebook.

2 Sample Input / Output

Here's an example query which I used to test the parser that I wrote. These should also be easily explainable by the LLMs, ChatGPT and Orca.

```
query1 = """SELECT * FROM students WHERE id = 1, name = Krish
GROUP BY class ORDER BY id"""
```

The parser response for this query would be

```
The given query selects all columns from the students table.
The query filters on the conditions: id = 1, name = Krish.
The query groups on the class column.
The query orders on the id column.
```

The local LLM gives the following response for this query

```
This query will retrieve all the information from
the table 'students' where the 'id' column is equal to 1
and the 'name' column is equal to 'Krish'
```

The local LLM actually misses the fact that there is a group by and order by clause in this query. It's possible that it has been trained on data including SELECT statements with WHERE clauses, but not including the GROUP and ORDER clauses. ChatGPT is able to explain this query in detail, and it even picks up on the fact that the ORDER BY clause could be missing the DESC or ASC keywords.

3 Metrics

Running the ChatGPT queries typically takes between 20 and 30 seconds each according to the timers in Jupyter Notebook. The local LLM queries take between 1 and 2 seconds. Using timeit, the parser I wrote takes roughly 100 microseconds to parse and explain each SQL statements. Making an API call to ChatGPT should take the longest since it needs to go over the network and wait for the request to be serviced. The local LLM is much faster but still takes at least a second since there is a need to run the query through the deep learning architecture and produce an output. The query parser I wrote is much faster since it relies on the structure of the query itself. The parser is less flexible than the LLM, since it expects a well formed query and only operates on a subset of queries. From a performance perspective, it seems that parsing is the way to go. I'm assuming that the parser could be generalized to more complex queries although that could be quite challenging.

4 Lessons Learned / Challenges

After experimentation, I realized that LLMs may not be well suited for explaining queries. Since SQL queries have a well defined structure, it is possible to write a parser which can decompose the clauses in a query. OpenAI's instance of ChatGPT seems quite capable of explaining simple SQL queries, but I'm unsure of its soundness on complex queries. If a query is far different from anything in the training set, ChatGPT will likely produce a response which sounds correct but ultimately is not. The parser approach seems to be more sound and executes faster than LLMs. However, it might be difficult in practice to write a parser which can succinctly describe a complex, nested query.

References

[1] Query plan
https://en.wikipedia.org/wiki/Query_plan

[2] Query optimization
https://en.wikipedia.org/wiki/Query_optimization

[3] Static code analysis
https://en.wikipedia.org/wiki/Static_program_analysis

[4] MySQL explain
https://dev.mysql.com/doc/refman/8.0/en/explain.html

[5] Adrenaline, static code analysis for Github repos
https://github.com/shobrook/adrenaline

[6] GPT4All reference
https://docs.gpt4all.io/gpt4all_python.html#chatting-with-gpt4all

[7] EvaDB Docs
https://evadb.readthedocs.io/en/stable/