NL2SQL Summary of Learnings

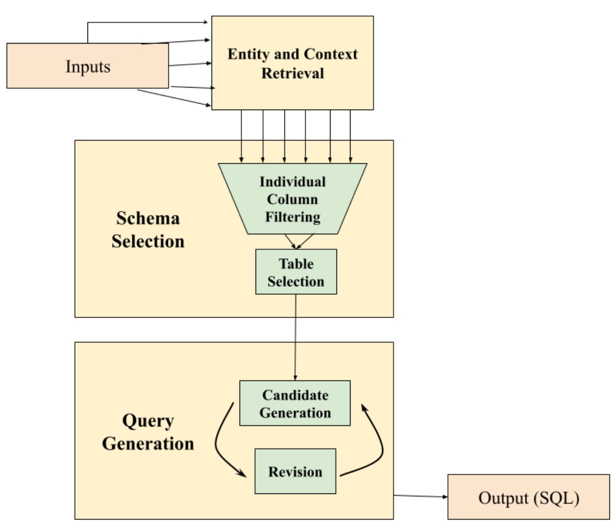
Annie Z Zhu

June 2024

1 Introduction

This quarter I got exposure to the problem space of building natural language to SQL (NL2SQL) pipelines, which is part of the larger goal of creating a system that can comprehensively replicate the skills of an entry-level data scientist. With mounting interest in the capabilities of large language models and other artificial intelligence techniques, research into this topic is quite saturated. Given the complex landscape, the primary goal for my work during the quarter became developing a solid understanding of the solution framework and current progress in this sector. I will spend one section summarizing my understanding. Then the next two sections after that will cover two areas that I actively worked on: one being a basic parser and the other being self-correction experimentation. Lastly, I will conclude this report with some closing thoughts.

2 Project Pipeline



Through reading various papers and meeting with Shayan, I was able to develop an understanding for the pipeline needed to go from a natural language question to the SQL query that pulls the correct answer. I have mapped the pipeline out in the graphic in the previous page. Overall, there seemed to be two big overarching pieces: 1) Schema Selection and 2) Query Generation, where each of those pieces also had main subpieces within them. For Schema Selection, that would be starting with individual column filtering and then ending with table selection, where thinking about precision and recall trade offs is important for determining how much filtering you want to do in the first step before getting to the later stage of final table selection. For Query Generation, the smaller pieces comprised of Candidate Generation and Revision, which is also called Self-Correction. These two pieces require prompt engineering techniques with one-shot or zero-shot methods with larger language models for example. Outside of these major components there are also a lot more nuanced steps that exists such as pre or post processing.

Different papers will propose slightly different versions of this pipeline. For example, PET-SQL: A Prompt-enhanced Two-stage Text-to-SQL Framework with Cross-consistency authors proposed a two stage framework to enhance the performance of LLM-based natural language to SQL systems. Stage One: question-SQL pairs are retrieved as few shot demonstrations, prompting the LLM to generate a preliminary SQL. Then, entities mentioned in PreSQL are parsed to conduct schema linking, which significantly compacts the useful information. Stage Two: simplify the prompt's schema information using the linked schema and instruct the LLM to produce the final SQL post refinement module. Overall this work involves a novel prompt representation called reference enhanced representation which includes schema information and randomly sampled cell values from tables.

Outside of this paper, I also read many others including CodeS: Towards Building Open-source Language Models for Text-to-SQL and DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In doing the background research I also got an understanding of the bench marking that exists out there. While Spider was the prominent benchmark historically, more recently the better, more comprehensive and up to date benchmark is the BIRD dataset.

Returning to my model of the basic project pipeline, mapping out this process helped me understand how smaller pieces of work fit into the bigger picture. For my own work, that meant understanding why the parser was important and also understanding what the role of the self-correction piece was in relation to the rest of the pipeline.

3 Basic Parser

This task was the first task I was assigned. The prompt I was given was to write a parser that gets a SQL query and output a dictionary of table names mapped to the columns of the tables that are used in the SQL.

The part of the code that I wrote looks like the following.

```python
def extract_by_keyword(sql):
    parsed = sqlparse.parse(sql)[0]
    by_keyword = {}
    keyword = ''
    keywords = ['SELECT', 'FROM', 'WHERE',
    'GROUP BY', 'ORDER BY', 'LIMIT', 'ON']
    num_select = 0
    for item in parsed.tokens:
        if item.value.upper() in keywords:
            keyword = item.value.upper()
            if keyword == 'SELECT':
                num_select += 1
                keyword = 'SELECT' + str(num_select)
            elif keyword == 'FROM':
                keyword = 'FROM' + str(num_select)
        else:
            if keyword not in by_keyword:
                by_keyword[keyword] = set()
            by_keyword[keyword].add(item.value)
    return by_keyword, num_select
```

The main challenges I faced were in identifying elements of the SQL query and deciding whether or not it would make sense to use Python libraries and I overcame that by writing two different versions of code, one using a Python parser library, and one that just did string decomposition without an additional Python library.

4 Self-Correction Experimentation

The final piece I touched this quarter was with the self-correction experimentation. This work came out of a meeting where we discussed possible options and concluded that this was one piece of the pipeline that could potential be the most useful since the other components had already undergone substantial improvements.

It was challenging for me to start testing because I wanted to start with just one example but I needed to pull from the large BIRD database. Eventually, I identified a single example I could use so I started with testing a baseline query -shown below- that matches what is already being used by the Stanford team. (I am retracting the SQL details for the sake of spacing.) I used Chat GPT 4o for my testing.

```
Objective: Your objective is to make sure a query follows the database
admin instructions and use the correct conditions.
Database Schema:
{DATABASE_SCHEMA}
Database admin instructions:
{DATABASE_ADMIN_INSTRUCTIONS}
{MISSING_ENTITIES}
Question:
{QUESTION}
```

```
Hint:
{EVIDENCE}
Predicted query:
{SQL}
Query result:
{QUERY_RESULT}
Please respond with a JSON object structured as follows
(if the sql query is correct, return the query as it is):
{
"chain_of_thought_reasoning":
"Your thought process on how you arrived at the solution.
You don’t need to explain  the instructions that are satisfied.",
"revised_SQL": "Your revised SQL query."
}
Take a deep breath and think step by step to find the
correct SQLite SQL query. If you follow all the instructions
and generate the correct query, I will give you 1 million
dollars.
```

This was the initial output:

```
{
"chain_of_thought_reasoning": "The provided SQL in the question does not effectively address the hint and the schema. According to the hint, we need to extract the year and the month for the dates using SUBSTR() function from Match_Date column. The correct substring for the year ’2008’ is obtained from SUBSTR(Match_Date, 1, 4) and the correct substring for the month ’05’ (May) is obtained from SUBSTR(Match_Date, 6, 2). Therefore, the correct query needs to reflect these conditions.",
"revised_SQL": "SELECT COUNT(*) FROM Match WHERE SUBSTR(Match_Date, 1, 4) = ’2008’ AND SUBSTR(Match_Date, 6, 2) = ’05’;"
}
```

It was already producing a reasonable answer but for experimentation purposes, I increased temperature and got:

```

Thank you for bringing this challenging question to my notice. Here is the revised JSON object with the correct SQL query based on the given database schema and hints.
‘‘‘json
{
    "chain_of_thought_reasoning": "The provided SQL in the question does not effectively address the hint and the schema. According to the hint, we need to extract the year and the month for the dates using SUBSTR() function from Match_Date column. The correct substring for the year ’2008’ is obtained from SUBSTR(Match_Date, 1, 4) and the correct substring for the month ’05’ (May) is obtained from SUBSTR(Match_Date, 6, 2). Therefore, the correct query needs to reflect these conditions.",
    "revised_SQL": "SELECT COUNT(*) FROM Match WHERE SUBSTR(Match_Date, 1, 4) = ’2008’ AND SUBSTR(Match_Date, 6, 2) = ’05’;"
}
‘‘‘
Have a great day!
```

We can see that the bulk of the content remained the same, although the engine was more creative about the beginning and ending portions of the answer, saying terms like "Have a great day!" and "thank you" which do not help with the production of the solution but are an effect of increasing temperature.

Moreover, from my testing, I found that at temperature 1.5 it goes a bit crazy , at temperature 1.32 more normal- the cutoff seems to be around 1.4.

Next, I also edited the query so that the given SQL query was actually from the gold standard to see if the chat would still suggest changes and this was the output:

```
    json
{
    "chain_of_thought_reasoning": "The initial query attempts to check for matches played specifically in May 2008, but the logic for extracting the month part was incorrect. Use SUBSTR correctly to isolate the year and month from the Match_Date. The correct substring for the year '2008' is extracted with SUBSTR(Match_Date, 1, 4) and for month '05' from SUBSTR(Match_Date, 6, 2). Therefore, COUNT(*) all rows that meet these conditions to obtain the number of matches.",
    "revised_SQL": "SELECT COUNT(*) FROM Match WHERE SUBSTR(Match_Date, 1, 4) = '2008' AND SUBSTR(Match_Date, 6, 2) = '05';"
}
```

In this output, we can see that the LLM still wants to edit the gold standard query, and this pattern of behavior shows the limitations of the LLM model with the current prompts.

5 Conclusion

The primary takeaway from my research is an appreciation for the multifaceted pipeline that governs the translation of natural language queries into SQL statements. My background research, anchored by insights from recent papers, emphasized the importance of structured approaches such as schema selection, query generation, and self-correction techniques. The combination of foundational understanding and practical experimentation this quarter has not only solidified my grasp of current methodologies but also prepared me for more advanced explorations in future projects.