From English to SQL: Schema-Aware Prompting with Large Language Models

Tingkai (Tank) Liu

https://www.linkedin.com/in/tankliu/

Abstract

In the era of data-driven decision-making, natural language interfaces for databases, commonly known as Text-to-SQL, are revolutionizing how analysts and business users interact with data. This article explores the innovative technique of schema-aware prompting with large language models (LLMs) like GPT-4, enabling the translation of plain English questions into executable SQL queries. By focusing on a pure prompt engineering approach, this method offers a lightweight and straightforward solution for rapid prototyping and integration into existing workflows. The article demonstrates the process using a sample e-commerce database, highlighting the importance of reverse-engineering database schemas and constructing focused prompts to enhance query accuracy. Additionally, it presents a practical use case where users can "chat" with their database, leveraging Python and ODBC to execute queries and return results in real-time. The pros and cons of schema-aware prompting are discussed, providing insights into its potential and limitations. Overall, this article showcases the transformative power of LLMs in creating intuitive and effective natural language interfaces for databases, paving the way for more accessible and insightful data analysis.

Introduction

Natural language interfaces for databases—commonly referred to as *Text-to-SQL*—are transforming how analysts and business users interact with data. Thanks to the capabilities of large language models (LLMs) like ChatGPT and Claude (*Vaswani et al, 2017*), it's now possible to translate plain English questions into executable SQL queries. This becomes especially powerful when the model has a clear understanding of the underlying database schema (*Radford et al, 2019*).

Multiple approaches of *Text-to-SQL* has been published (*Hwang, 2019; Heng, 2020; Jichuan 2020*). All these approach seems complicated to ordinary LLM users. People are likely to think that SQL agent tools may help. However, the integration of such agents complicates the problem we really need to resolve in a lot of cases. It requires additional investments to buy the agent tools, additional resource to implement it and additional controls in place to avoid internal data leak out to third party. There is a need to leverage pure *prompt engineering* techniques to achieve *Text-to-SQL* goal.

In this article, I'll demonstrate a pure *prompt engineering* technique for schema-aware SQL generation using a sample e-commerce database. Unlike SQL agents, which require additional tooling and infrastructure, this approach is lightweight and straightforward—ideal for rapid prototyping or integrating into existing workflows.

Materials & Methods

The LLM model I used is ChatGPT-4o with training data cutoff date of October 2023. ChatGPT and other LLMs will also work. No additional materials are required in this application.

Here are the methods I used.

Step 1: Reverse-Engineering Your Database Schema

To generate accurate SQL, the LLM needs to understand the structure of your data. But you don't need to connect to the actual database or dump the entire schema. Instead, you can provide a simplified and focused version of the schema using CREATE TABLE statements, tailored to the specific tables or views relevant to your use case.

Even if you're querying a view, treat it like a table in your prompt to simplify the task for the model. You can start from a SELECT statement and ask the LLM to reverse-engineer it into a CREATE TABLE definition with annotated comments. Here's an example prompt:

Prompt:
Reverse engineer the following SQL SELECT statement into a CREATE TABLE statement with comments:

```
SELECT
  cid AS customer_id,      -- unique customer identifier (primary key)
  fn  AS first_name,       -- customer's first name
  ln  AS last_name,        -- customer's last name
  em  AS email,            -- customer's email address
  ct  AS city,             -- customer's city
  rg  AS region            -- customer's geographical region
FROM cust;
```

Why Focus on Only What You Need?

Enterprise databases often contain thousands of tables, but most teams only work with a handful of them. When constructing your prompt template, include only the necessary tables, fields, and join logic that are relevant to your business function.

Avoid using tools that auto-extract full schemas from the database, as this introduces unnecessary complexity and noise. A focused schema allows the model to reason more effectively and generate concise, accurate SQL queries.
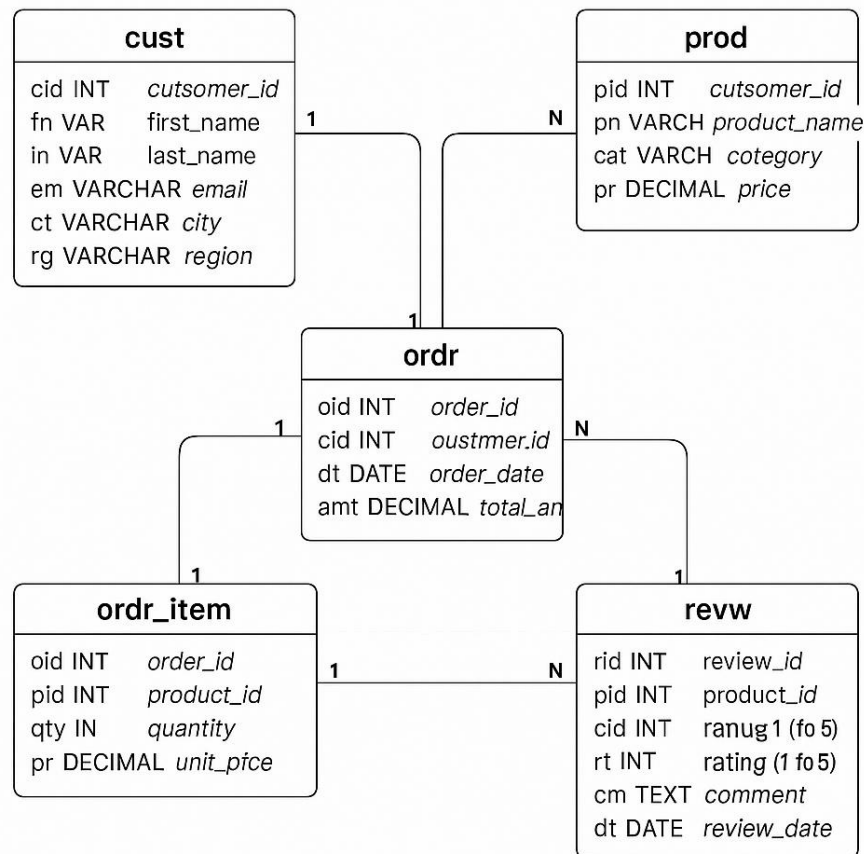
Fig. 1 ER diagram for the sample database

Fig. 1 is a sample database schema. We'll work with this compact database containing five tables in this article. Column names are abbreviated, and each includes a comment for clarity. Here is the ER diagram of the database.

Step 2: Make the LLM Aware of Your Schema

To help the LLM understand the structure, include the table and field information in your prompt. Here's a reusable prompt template:

> **Prompt:**
> *You are a SQL assistant that helps generate correct and optimized SQL queries based on natural language questions. Use the standard SQL style that should compatible with PostgreSQL and AWS Redshift.*
>
> *The database schema is as follows:*
> *[The schema is with table names, abbreviated field names, and inline comments]*

```
-- Table: cust (Customers)
CREATE TABLE cust (
  cid INT PRIMARY KEY,            -- customer_id
  fn VARCHAR(50),                 -- first_name
  ln VARCHAR(50),                 -- last_name
  em VARCHAR(100),                -- email
  ct VARCHAR(50),                 -- city
  rg VARCHAR(50)                  -- region
);

-- Table: prod (Products)
CREATE TABLE prod (
  pid INT PRIMARY KEY,            -- product_id
  pn VARCHAR(100),                -- product_name
  cat VARCHAR(50),                -- category
  pr DECIMAL(10, 2)               -- price
);

-- Table: ordr (Orders)
CREATE TABLE ordr (
  oid INT PRIMARY KEY,            -- order_id
  cid INT,                        -- customer_id
  dt DATE,                        -- order_date
  amt DECIMAL(10, 2),             -- total_amount
  FOREIGN KEY (cid) REFERENCES cust(cid)
);

-- Table: ordr_item (Order Items)
```

```
CREATE TABLE ordr_item (
  oid INT,                        -- order_id
  pid INT,                        -- product_id
  qty INT,                        -- quantity
  pr DECIMAL(10, 2),              -- unit_price
  FOREIGN KEY (oid) REFERENCES ordr(oid),
  FOREIGN KEY (pid) REFERENCES prod(pid)
);

-- Table: revw (Reviews)
CREATE TABLE revw (
  rid INT PRIMARY KEY,            -- review_id
  pid INT,                        -- product_id
  cid INT,                        -- customer_id
  rt INT,                         -- rating (1 to 5)
  cm TEXT,                        -- comment
  dt DATE,                        -- review_date
  FOREIGN KEY (pid) REFERENCES prod(pid),
  FOREIGN KEY (cid) REFERENCES cust(cid)
);
```

Here are sample examples of how I want to filter data of my dimensions. Whenever a dimension value is mentioned, help me automatically associate the data field to create the filter. For example, if I mention midwest, you should know I am talking about rg in ('Midwest').

Filter 1: the cities of customer.
```
Select count(1) from cust where ct in ('Chicago', 'New York','Austin',
'San Francisco');
```
Filter 2: the regions of customer.
```
Select count(1) from cust where rg in ('Midewest','East','South','West');
```
Filter 3: the product category.
```
Select count(1) from prod where cat in
('Electronics','Footwear','Fitness');
```

Here are some sample questions for you to learn how I want the output of your answers:

Q1: List the top 5 products with the highest average rating and include their category.
A1:
```
SELECT p.pn, p.cat, AVG(r.rt) AS avg_rating
FROM prod p
JOIN revw r ON p.pid = r.pid
GROUP BY p.pn, p.cat
ORDER BY avg_rating DESC
LIMIT 5;
```
Q2: Show total sales (amount) per product category.
A2:
```
SELECT p.cat, SUM(oi.qty * oi.pr) AS total_sales
FROM prod p
JOIN ordr_item oi ON p.pid = oi.pid
```

```
GROUP BY p.cat;
```

Q3: List customers from the city of 'Chicago' who rated a product 5 stars.

A3:

```
SELECT DISTINCT c.fn, c.ln, c.em
FROM cust c
JOIN revw r ON c.cid = r.cid
WHERE c.ct = 'Chicago' AND r.rt = 5;
```

Q4: What are the top 3 customers who spent the most money in the last 6 months?

A4:

```
SELECT c.fn, c.ln, SUM(o.amt) AS total_spent
FROM cust c
JOIN ordr o ON c.cid = o.cid
WHERE o.dt >= CURRENT_DATE - INTERVAL '6 months'
GROUP BY c.fn, c.ln
ORDER BY total_spent DESC
LIMIT 3;
```

Q5: Show the average rating per category for products that cost more than $100.

A5:

```
SELECT p.cat, AVG(r.rt) AS avg_rating
FROM prod p
JOIN revw r ON p.pid = r.pid
WHERE p.pr > 100
GROUP BY p.cat;
```

Instructions:

Whenever a question is asked, generate an SQL for it, and prompt me to ask another question until I type 'Exit()'. If I type 'Another SQL question', re-enter the SQL generation context.

Here is my first question:

What is the most order product in Midwest?

Results & Discussion

After I paste the prompt into LLM, I tested out below questions and got corresponding SQL.

Q1: What are the top 5 products of Footwear by sales sold in Boston of Q2 '25?

Result SQL:

```
SELECT p.pn, SUM(oi.qty * oi.pr) AS total_sales
FROM prod p
JOIN ordr_item oi ON p.pid = oi.pid
JOIN ordr o ON oi.oid = o.oid
JOIN cust c ON o.cid = c.cid
WHERE c.ct = 'Boston' AND p.cat = 'Footwear' AND o.dt BETWEEN '2025-04-
01' AND '2025-06-30'
```

```
GROUP BY p.pn
ORDER BY total_sales DESC
LIMIT 5;
```

Q2: Find the most recent review for each product.

Result SQL:
```
SELECT p.pn, r.cm, r.rt, r.dt
FROM prod p
JOIN revw r ON p.pid = r.pid
WHERE r.dt = (
    SELECT MAX(r2.dt)
    FROM revw r2
    WHERE r2.pid = r.pid
)
ORDER BY p.pn;
```

Q3: List the orders where the total order amount is inconsistent with the total of order items.

Result SQL:
```
SELECT o.oid, o.amt, SUM(oi.qty * oi.pr) AS calculated_total
FROM ordr o
JOIN ordr_item oi ON o.oid = oi.oid
GROUP BY o.oid, o.amt
HAVING o.amt <> SUM(oi.qty * oi.pr);
```

Q4: Find the average rating of products in the 'Electronics' category that were ordered by customers from 'New York' in the last year.

Result SQL:
```
SELECT AVG(r.rt) AS avg_rating
FROM prod p
JOIN revw r ON p.pid = r.pid
JOIN ordr_item oi ON p.pid = oi.pid
JOIN ordr o ON oi.oid = o.oid
JOIN cust c ON o.cid = c.cid
WHERE p.cat = 'Electronics' AND c.ct = 'New York' AND o.dt >=
CURRENT_DATE - INTERVAL '1 year';
```

Q5: List the top 3 regions by total sales amount, including the number of orders and average order amount.

Result SQL:
```
SELECT c.rg, COUNT(o.oid) AS num_orders, SUM(o.amt) AS total_sales,
AVG(o.amt) AS avg_order_amount
FROM cust c
JOIN ordr o ON c.cid = o.cid
GROUP BY c.rg
```

```
ORDER BY total_sales DESC
LIMIT 3;
```

Q6: Show the top 5 customers who have given the most reviews, along with their average rating.

Result SQL:
```
SELECT c.fn, c.ln, COUNT(r.rid) AS num_reviews, AVG(r.rt) AS avg_rating
FROM cust c
JOIN revw r ON c.cid = r.cid
GROUP BY c.fn, c.ln
ORDER BY num_reviews DESC
LIMIT 5;
```

Q7: Find the total sales and average rating for products that have been ordered more than 100 times.

Result SQL:
```
SELECT p.pn, SUM(oi.qty * oi.pr) AS total_sales, AVG(r.rt) AS avg_rating
FROM prod p
JOIN ordr_item oi ON p.pid = oi.pid
JOIN revw r ON p.pid = r.pid
GROUP BY p.pn
HAVING SUM(oi.qty) > 100;
```

Q8: List the top 5 cities by the number of unique customers who have placed orders in the last 3 months.

Result SQL:
```
SELECT c.ct AS city, COUNT(DISTINCT c.cid) AS unique_customers
FROM cust c
JOIN ordr o ON c.cid = o.cid
WHERE o.dt >= CURRENT_DATE - INTERVAL '3 months'
GROUP BY c.ct
ORDER BY unique_customers DESC
LIMIT 5;
```

Q9: Find the top 3 products with the highest total sales amount that have received at least 20 reviews.

Result SQL:
```
SELECT p.pn, SUM(oi.qty * oi.pr) AS total_sales
FROM prod p
JOIN ordr_item oi ON p.pid = oi.pid
JOIN revw r ON p.pid = r.pid
GROUP BY p.pn
HAVING COUNT(r.rid) >= 20
ORDER BY total_sales DESC
LIMIT 3;
```

Q10: Identify the top 5 customers who have spent the most money on 'Electronics' products in the last year. For each of these customers, provide their full name, email, total amount spent, the number of orders they placed, and the average rating they gave to 'Electronics' products. Additionally, include the most frequently purchased 'Electronics' product by each customer.

Result SQL:
```
WITH CustomerElectronicsSpending AS (
    SELECT c.cid, c.fn, c.ln, c.em, SUM(oi.qty * oi.pr) AS total_spent,
COUNT(DISTINCT o.oid) AS num_orders
    FROM cust c
    JOIN ordr o ON c.cid = o.cid
    JOIN ordr_item oi ON o.oid = oi.oid
    JOIN prod p ON oi.pid = p.pid
    WHERE p.cat = 'Electronics' AND o.dt >= CURRENT_DATE - INTERVAL '1
year'
    GROUP BY c.cid, c.fn, c.ln, c.em
),
CustomerElectronicsRating AS (
    SELECT r.cid, AVG(r.rt) AS avg_rating
    FROM revw r
    JOIN prod p ON r.pid = p.pid
    WHERE p.cat = 'Electronics' AND r.dt >= CURRENT_DATE - INTERVAL '1
year'
    GROUP BY r.cid
),
CustomerFrequentProduct AS (
    SELECT oi.cid, p.pn, COUNT(oi.pid) AS purchase_count
    FROM ordr_item oi
    JOIN prod p ON oi.pid = p.pid
    WHERE p.cat = 'Electronics'
    GROUP BY oi.cid, p.pn
    ORDER BY oi.cid, purchase_count DESC
)

SELECT ces.fn, ces.ln, ces.em, ces.total_spent, ces.num_orders,
cer.avg_rating, cfp.pn AS most_frequent_product
FROM CustomerElectronicsSpending ces
LEFT JOIN CustomerElectronicsRating cer ON ces.cid = cer.cid
LEFT JOIN (
    SELECT DISTINCT ON (cfp.cid) cfp.cid, cfp.pn
    FROM CustomerFrequentProduct cfp
    ORDER BY cfp.cid, cfp.purchase_count DESC
) cfp ON ces.cid = cfp.cid
ORDER BY ces.total_spent DESC
LIMIT 5;
```

In the above testing questions, LLM is able to generate high quality SQLs to answer natural language questions. This article is not intended to research on how accurate it generate SQL. Instead it intends to provide a practical way for LLM users to quickly implement Text-to-

SQL technique in their workflow. The quality of the resulting SQL depends on the quality of the prompt and the LLM that is used.

The Pro and Cons of Schema-Aware Prompting

Schema-aware prompting offers several advantages and disadvantages when it comes to generating SQL queries using large language models (LLMs):
Pros:

1. **Precision and Accuracy**: By providing a clear and focused schema, the LLM can generate more precise and accurate SQL queries. This reduces the likelihood of errors and ensures that the queries align with the intended data structure.
2. **Efficiency**: Schema-aware prompting streamlines the query generation process by focusing only on relevant tables and fields. This minimizes unnecessary complexity and allows for faster query generation.
3. **Ease of Integration**: This approach is lightweight and can be easily integrated into existing workflows without the need for additional tooling or infrastructure. It is ideal for rapid prototyping and quick iterations.
4. **Improved Reasoning**: With a clear understanding of the schema, the LLM can better reason about the relationships between tables and fields, leading to more insightful and optimized queries.

Cons:

1. **Manual Schema Definition**: The process requires manual effort to define and provide the schema to the LLM. This can be time-consuming, especially for large databases with complex structures.
2. **Dependency on Schema Accuracy**: The effectiveness of this approach relies heavily on the accuracy of the provided schema. Any discrepancies or errors in the schema can lead to incorrect query generation.
3. **Potential for Over-Simplification**: By focusing only on relevant tables and fields, there is a risk of oversimplifying the schema, which may overlook important data relationships or constraints.

Use Case: Chat with Your Database

Schema-aware prompting with LLMs opens up exciting possibilities for creating interactive and conversational interfaces with databases. One such use case is enabling users to "chat" with their database, where they can ask questions in natural language and receive real-time answers. Here's how this can be achieved:
1. **Natural Language Querying**: Users can input questions in plain English, such as "What are the top-selling products in the Midwest?" or "Show me the total sales for

electronics last month." The LLM processes these questions and generates corresponding SQL queries.

2. **SQL Execution via Python and ODBC**: Once the SQL query is generated, it can be executed using Python. By leveraging ODBC (Open Database Connectivity), Python can establish a connection to the database, execute the SQL query, and retrieve the results.

3. **Returning Results to the LLM**: After executing the query and obtaining the results, the data can be formatted and returned to the LLM. The LLM can then present the results to the end user in a conversational manner, providing insights and answers to their questions.

4. **Interactive and Dynamic Experience**: This setup allows users to interact with their database in a dynamic and conversational way. They can ask follow-up questions, refine their queries, and explore data without needing to write complex SQL themselves.

By integrating schema-aware prompting with Python and ODBC, businesses can create powerful natural language interfaces that enhance data accessibility and empower users to make data-driven decisions more intuitively. This use case demonstrates the potential of LLMs to transform traditional database interactions into engaging and user-friendly experiences.

Conclusion

Schema-aware prompting with LLMs represents a powerful technique for generating SQL queries from natural language questions. By leveraging the capabilities of LLMs and providing a focused schema, users can achieve precise and efficient query generation. While this approach offers significant advantages in terms of accuracy and ease of integration, it also requires careful consideration of schema definition and may have limitations in context size. Overall, schema-aware prompting is a valuable tool for transforming how analysts and business users interact with data, enabling more intuitive and effective data analysis.

Reference
1. **Vaswani, A., et al. (2017).** Attention is All You Need. In Advances in Neural Information Processing Systems (NeurIPS).
2. **Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019).** Language Models are Unsupervised Multitask Learners. OpenAI Blog.
3. **Hwang, W., Yim, J., Park, S., & Seo, M. (2019).** A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization. arXiv preprint arXiv:1902.01069.
4. **Heng G. et al. (2020)** Few-shot Table-to-Text Generation with Table Structure Reconstruction and Content Matching. In Proceedings of the 28th International Conference on Computational Linguistics, pages 1978–1988, Barcelona, Spain (Online).

5. **Jichuan Z., et al. (2020).** Photon: A Robust Cross-Domain Text-to-SQL System. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 204–214, Online.