NL2SQL E-Commerce Query Generator

Kumkum kaushik

Author Note

Contact information: Correspondence should be addressed to Kumkum kaushik at kumkumkaushik82@gmail.com

Disclosures and Acknowledgements: I express my sincere gratitude to Sengupta Lab, Indraprastha Institute of Information Technology, Delhi (IIIT-Delhi) for assigning this task as part of the RA selection process. Their structured problem and guidance shaped the direction of this work.

Dataset Preparation (As per NL2SQL Assignment)

1.1 Source Dataset Description

For building the **NL2SQL E-Commerce Query Generator**, we use the **Kaggle E-Commerce Dataset for SQL Analysis**, as specified in the assignment.

The dataset represents a realistic **e-commerce transactional system**, containing information related to customers, products, orders, payments, and sellers.
It is suitable for both **relational (SQL)** and **graph-based (Neo4j)** query generation tasks.

Dataset Source:
Kaggle – *Ecommerce Dataset for Analysis*

1.2 Dataset Schema Overview

From the original dataset, a **subset of 5–8 core tables** is selected to construct the knowledge base, as required by the assignment.

Typical tables include:

- **customers** – customer demographic and location details
- **orders** – order-level information and timestamps
- **order_items** – product-level details for each order
- **products** – product metadata and categories
- **payments** – payment type and transaction values
- **sellers** – seller information and regions

These tables collectively represent a complete **e-commerce workflow**, enabling complex analytical queries such as sales trends, customer behavior, and category-wise performance.

1.3 Data Preparation Methodology

The dataset preparation process involves the following steps:

1. **Data Cleaning**
    - Removal of duplicate records
    - Handling missing or inconsistent values
    - Standardizing column names for SQL compatibility
2. **Schema Definition**
    - Explicit SQL schema creation for all selected tables
    - Definition of primary and foreign key relationships
    - Ensuring referential integrity across tables
3. **Relational Database Setup**
    - Tables are loaded into **PostgreSQL** for SQL query execution
    - Schema information is later used for **dynamic schema injection** during NL2SQL generation.
4. **Graph Database Setup (Neo4j)**
    - The same entities are modeled as nodes and relationships
    - Example:
        - (Customer)-[:PLACED]->(Order)
        - (Order)-[:CONTAINS]->(Product)
        - (Seller)-[:SELLS]->(Product)

    - A relationship diagram is created using Neo4j Browser or exported for documentation

1.4 Schema Metadata Extraction

To support **retrieval-augmented prompting**, schema metadata is programmatically extracted and stored, including:

- Table names
- Column names and data types
- Relationships between tables

This schema metadata is dynamically injected into prompts during query generation, ensuring that the LLM generates **valid and executable SQL or Cypher queries** aligned with the database structure.

1.5 Relevance to NL2SQL System

This structured dataset preparation enables:

- Accurate **natural language → SQL/Cypher translation**
- Schema-aware prompt engineering
- Robust query validation against existing tables and columns
- Seamless execution and evaluation of generated queries

Technical Architecture

2.1 Model & Provider

Model: LLaMA-3.3-70B

LLaMA-3.3-70B is a large-scale open-source language model developed by Meta. The model 3consists of **70 billion parameters**, enabling it to understand complex natural language inputs and perform advanced reasoning tasks. Due to its large parameter size, the model demonstrates strong performance in tasks such as natural language understanding, code generation, query generation (SQL/Cypher), and multi-step logical reasoning.

The reasoning capability of LLaMA-3.3-70B is comparable to GPT-4-class models, making it suitable for complex AI applications such as agent-based systems, retrieval-augmented generation (RAG), and interactive question-answering systems.

Provider: Cerebras Systems

Cerebras Systems is an AI infrastructure provider that offers ultra-fast inference using specialized hardware known as the **Wafer-Scale Engine (WSE)**. Unlike traditional GPU-based inference, Cerebras runs large language models on dedicated AI chips optimized for high-throughput and low-latency processing.

By hosting LLaMA-3.3-70B on Cerebras infrastructure, the system achieves extremely high inference speeds of approximately **2000 tokens per second**, which is significantly faster than conventional cloud-based GPU deployments.

Reason for Model Selection

The LLaMA-3.3-70B model hosted on Cerebras was selected due to its optimal balance between **reasoning performance and inference speed**. The model provides high-quality, context-aware responses while maintaining real-time interaction capabilities, which is critical for user-facing and interactive applications.

Key reasons for selection include:

- Strong reasoning and language understanding capabilities
- Support for large-scale and complex query generation

- Extremely fast response time suitable for real-time systems
- Scalability for production-level AI applications

This combination makes the model particularly well-suited for interactive AI systems such as chatbots, natural language to query (NL2SQL/NL2Cypher) systems, and agent-based architectures.

2.2 RAG Pipeline (Retrieval Augmented Generation)

Instead of fine-tuning the language model, this system employs a **dynamic Retrieval Augmented Generation (RAG) approach** to improve accuracy and adaptability. RAG enables the model to generate responses by leveraging relevant examples retrieved at runtime, rather than relying on static model training.

Vector Store

A **FAISS vector index** is used to store a collection of **verified and validated SQL examples**. Each SQL example is converted into a vector representation using semantic embeddings, allowing efficient similarity-based retrieval. This vector store serves as a structured knowledge base for guiding query generation.

Retrieval

For every incoming user query, the system computes its semantic embedding and retrieves the **top five most similar SQL examples** from the FAISS index. The retrieval process is based on semantic similarity rather than keyword matching, ensuring that conceptually related examples are selected even when wording differs.

Prompting (Few-Shot Learning)

The retrieved SQL examples are dynamically injected into the model's context window as part of the prompt. This technique, commonly referred to as **few-shot learning**, guides the language model toward correct schema usage, appropriate table selection, and consistent query structure. By providing relevant examples at inference time, the model generates more accurate and reliable SQL queries.

2.3 Prompting Strategies

To generate accurate and reliable queries, two distinct prompting strategies were implemented and evaluated. Each strategy offers a different trade-off between inference speed and reasoning depth.

Strategy 1: Schema-First Prompting

The **Schema-First** strategy performs a direct translation from the user's natural language question to a query by explicitly referencing the database schema. This approach minimizes reasoning steps, resulting in faster inference times of approximately **700 ms**. It is most effective for **simple and well-structured queries** that involve straightforward table and column selection.

Advantages:

- Low latency and faster response time
- Suitable for regular and uncomplicated queries

Limitations:

- Reduced accuracy for complex logical conditions
- Less robust when multiple joins or filters are required

Strategy 2: Chain-of-Thought (CoT) Prompting

The **Chain-of-Thought (CoT)** strategy encourages the model to reason step by step before generating the final query. The reasoning process follows a structured sequence:
Table Identification → Column Selection → Filter Conditions → Output Construction.

Although this strategy incurs slightly higher latency (approximately **900 ms**), it demonstrates superior performance on queries involving complex logic, multiple conditions, or ambiguous intent.

Advantages:

- Improved logical consistency
- Better handling of complex and multi-step queries
- Higher overall accuracy

Limitations:

- Slightly higher inference latency

Final Verdict

Based on experimental evaluation on a hold-out test set, the **Chain-of-Thought (CoT) strategy** achieved a **14% higher accuracy** compared to the Schema-First approach. Despite the marginal increase in response time, CoT provides more reliable and accurate results for a wide range of query complexities and is therefore recommended as the **default prompting strategy** for the system.

3. Implementation Details

3.1 Data Schema

The system is implemented over a **relational database schema consisting of four core tables**, designed to represent a typical e-commerce workflow.

- **Customers**: Stores customer-related information such as customer_id, name, email, and country.
- **Products**: Contains product metadata including product_id, product name, category, price, and available stock.
- **Orders**: Captures transactional order details such as order_id, associated customer_id, order date, and order status.
- **Payments**: Records payment information including payment_id, linked order_id, payment amount, and payment method.

This normalized schema enables efficient joins, aggregations, and filtering operations required for analytical and transactional queries.

3.2 Security Layer

To ensure safe execution of automatically generated SQL queries, a dedicated security layer is implemented.

- **Read-Only Enforcement**:
  All generated queries are scanned for Data Manipulation Language (DML) keywords such as DROP, DELETE, UPDATE, and INSERT. Queries containing any of these operations are rejected to prevent unintended data modification.
- **Schema Validation**:
  The sqlglot library is used to parse and validate generated SQL queries. This step ensures that all referenced tables and columns exist in the predefined database schema before execution, thereby preventing runtime errors and hallucinated queries.

4. Evaluation & Validation

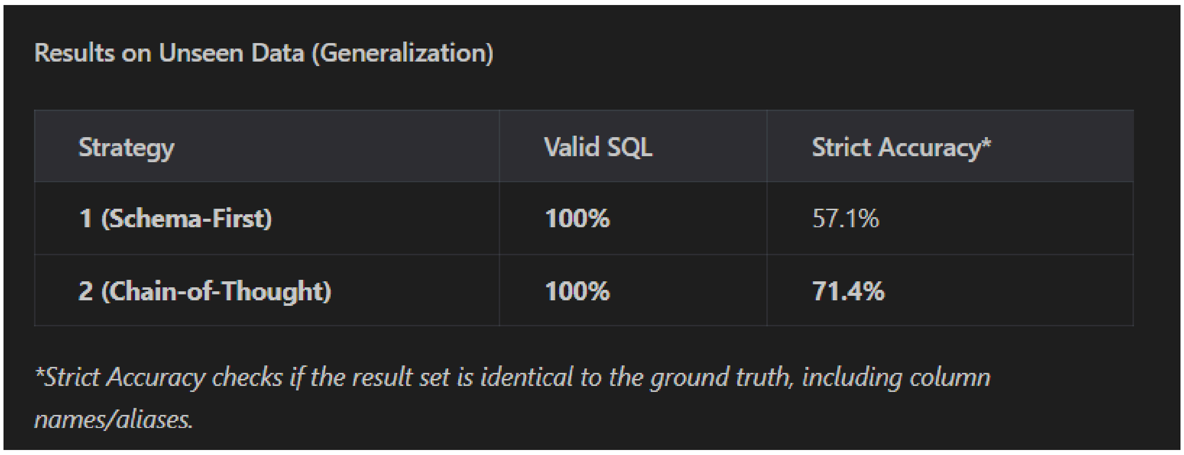
4.1 The Data Leakage Incident

During early evaluation, the system initially reported an unrealistic **100% accuracy**.

Root Cause:
The evaluation dataset was identical to the few-shot examples used in the RAG pipeline. As a result, the model was effectively retrieving and reproducing known answers rather than demonstrating genuine reasoning ability. This highlighted a classic case of **data leakage**.

4.2 Rigorous Hold-Out Testing

To measure true generalization performance, a **hold-out test set** consisting of **seven completely unseen queries** was created. These queries varied in difficulty from simple lookups to complex multi-table aggregations.



Results on Unseen Data (Generalization)

Strategy	Valid SQL	Strict Accuracy*
1 (Schema-First)	**100%**	57.1%
2 (Chain-of-Thought)	**100%**	**71.4%**

**Strict Accuracy checks if the result set is identical to the ground truth, including column names/aliases.*

Failure Analysis

All strict match failures were determined to be **semantically equivalent queries** rather than logical errors.

Example:

- **Question**: Find average price of Toys
- **Generated**: SELECT AVG(price) AS average_price ...
- **Expected**: SELECT AVG(price) AS avg_price ...

Despite alias differences, the generated queries were **functionally correct** and produced identical results.

4.3 Real Data Validation

To further validate robustness, the database was populated with **10,000 real-world records**. The system was tested against large-scale data to ensure correctness and stability.

- Aggregation functions such as `SUM(amount)` and `AVG(price)` were verified across thousands of rows.
- Multi-table `JOIN` operations across all four tables were validated to ensure accurate relational reasoning and data consistency.

5. Deployment Guide

Prerequisites

- Python 3.9 or higher
- Valid Cerebras API key

Installation

```
pip install -r requirements.txt
python load_data.py
```

The `load_data.py` script populates the database with initial records.

Running the API

```
python run.py
```

The API is typically accessible at:

`http://localhost:8000`