Natural Language to SQL (Text2SQL) Engine

Prepared by: Abhishek Gupta

1. Executive Summary

The objective of this assignment was to design and build a robust system that empowers users to query a structured database using natural language. The core challenge lies not only in accurately translating language to SQL but also in ensuring the generated queries are **safe, valid, and aligned with the database schema.**

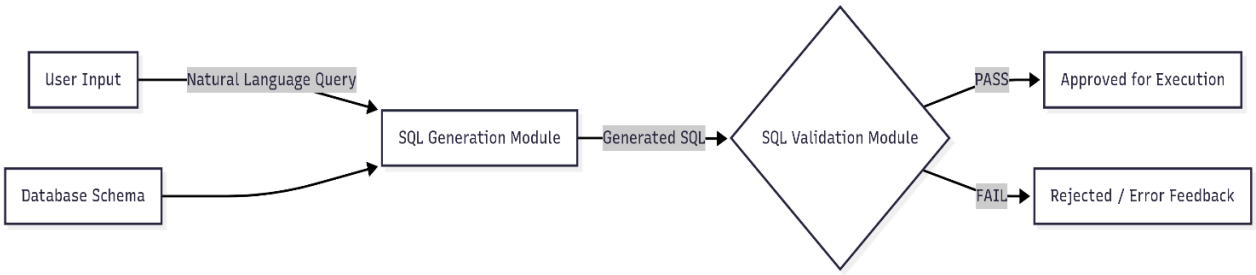
My solution is a **two-stage** that addresses both assignments directly.

1. **AI-Powered Generation:** A Large Language Model (Google's Gemini) is used to interpret the user's intent and generate a candidate SQL query.
2. **Rule-Based Validation:** A critical validation layer then inspects the generated SQL. This layer acts as a system "firewall," rejecting any query that is unsafe, syntactically incorrect, or references non-existent tables or columns.

This document outlines the system architecture, provides an in-depth explanation of each component along with executable code, and showcases the complete pipeline in action.

2. High-Level System Architecture

The system follows a clear, sequential flow from user input to a final, trusted SQL query.



1. **SQL Generation Module:** Takes the user's query and the database schema to produce a candidate SQL string.
2. **SQL Validation Module:** The system's security gate. It rigorously checks the candidate SQL against a set of rules.
3. **Final Decision:** The query is either approved for execution or rejected, with a clear reason for the failure.

3. Component Deep Dive: The AI-Powered SQL Generator

The intelligence of the system lies in its ability to understand language. This is achieved through a technique called **Prompt Engineering**.

Purpose: To create a detailed, context-rich instruction set (a "prompt") for the LLM that guides it to produce the correct SQL.

Implementation: A Python function dynamically combines three key pieces of information:

1. **The Task:** A clear directive telling the model it is an expert SQL developer.
2. **The Schema:** The complete, flattened database schema, so the model knows exactly what tables and columns are available.
3. **The User's Question:** The specific query to be translated.

The code below shows the function that builds this prompt.

```
def create_sql_prompt(schema_df, user_query):

  prompt = """You are an expert SQL developer. Your task is to write a SQL query based on the provided database schema and a user's question.

  Follow these rules strictly:

    1. Only use the tables and columns listed in the schema below.
    2. 2. If a user's question involves a name (like "John Doe"), you must perform a lookup on the 'users' table.
    3. The generated SQL must be syntactically correct for a standard SQL database.
    4. If the question cannot be answered with the given schema, respond with "Error: Cannot answer query with the provided schema."

  DATABASE SCHEMA:{schema}

  USER QUESTION:"{query}"

  SQL QUERY:

  """

  schema_string = schema_df.to_string()

  return prompt.format(schema=schema_string, query=user_query)
```

This approach correctly generated a complex query requiring a JOIN from a simple question.

4. Component Deep Dive: The Critical Validation Layer

Purpose: To enforce security and correctness. It ensures that only well-formed, safe, and schema-compliant queries can proceed.

Implementation: The SQLValidator class performs a sequence of checks. If any check fails, the process stops immediately.

1. **Safety Check:** Scans for blacklisted keywords (DELETE, UPDATE, DROP, etc.) to prevent any data modification.
2. **Syntactic & Semantic Check:** Uses a SQL parsing library to verify the query's structure and ensure every table and column referenced actually exists in our schema.

The code below defines the validator's structure.

```
class SQLValidator:
  def __init__(self, schema_df):
    self.schema_df = schema_df
    self.known_tables = set(schema_df['table_name'].unique())
```

```
        self.unsafe_keywords = {'DELETE', 'UPDATE', 'DROP', 'INSERT', 'TRUNCATE',
'GRANT', 'REVOKE'}
    def validate(self, sql_query):
        is_safe, message = self._is_safe(sql_query)
        if not is_safe:
            return False, message


        try:
            is_valid, message = self._are_tables_and_columns_valid(sql_query)
            if not is_valid:
                return False, message
        except Exception as e:
            return False, f"Syntactic Error: Failed to parse query. Details: {e}
        return True, "Validation successful: Query is safe and aligned with the schema."
```

5. The Integrated Pipeline in Action

This is the point where all components integrate to form the complete system. The text_to_sql_pipeline function coordinates the entire end-to-end workflow.

Demonstration 1: A Valid and Safe Query
The system correctly generates the SQL, the validator passes it, and the query is **APPROVED**.

1. Received User Query: 'List all open incidents reported by John Doe.'
2. Generated SQL from LLM:

```
SELECT i.*
FROM incidents i
JOIN users u ON i.user_id = u.user_id
WHERE u.name = 'John Doe' AND i.status = 'open'
```

3. Initializing SQL Validator...
4. Validation Result: PASS
5. FINAL DECISION: Query is APPROVED for execution.

Demonstration 2: An Unsafe Query
The LLM generates a query with a DELETE keyword. The validator's safety check immediately catches it, and the query is **REJECTED**.

1. Received User Query: 'Delete all logs related to the user 'admin'.'
2. Generated SQL from LLM:

```
DELETE FROM logs WHERE user_id = (SELECT user_id FROM users WHERE name = 'admin')
```

3. Initializing SQL Validator...
4. Validation Result: FAIL
5. FINAL DECISION: Query is REJECTED.

6. Trade-offs and Future Work

This prototype is robust, but for a production system, further enhancements could be considered:

- **Advanced Parsing:** Using a library that creates an Abstract Syntax Tree (AST) would allow for even deeper validation, such as checking data types in WHERE clauses.

- **Performance:** The LLM API call introduces latency.
- **User Feedback Loop:** Capturing rejected queries and enabling users to flag inaccurate translations would help build a valuable dataset for ongoing system improvement.

7. Conclusion

This project effectively showcases the design and implementation of a complete Natural Language to SQL (Text2SQL) system. By combining an AI-driven query generator with a robust validation layer, the solution delivers advanced natural language understanding while maintaining the security standards essential for enterprise use.
