# DBMS PROJECT

## Group Details :

Chepuri Naga Venkata Varsha - 21CS30014
Chitiyala RaviTeja - 21CS30016
Budagam Devichand - 21CS30012
Tadigoppala Sumanth - 21CS30053
Singabattu Sathya - 21CS10061

## Objective :

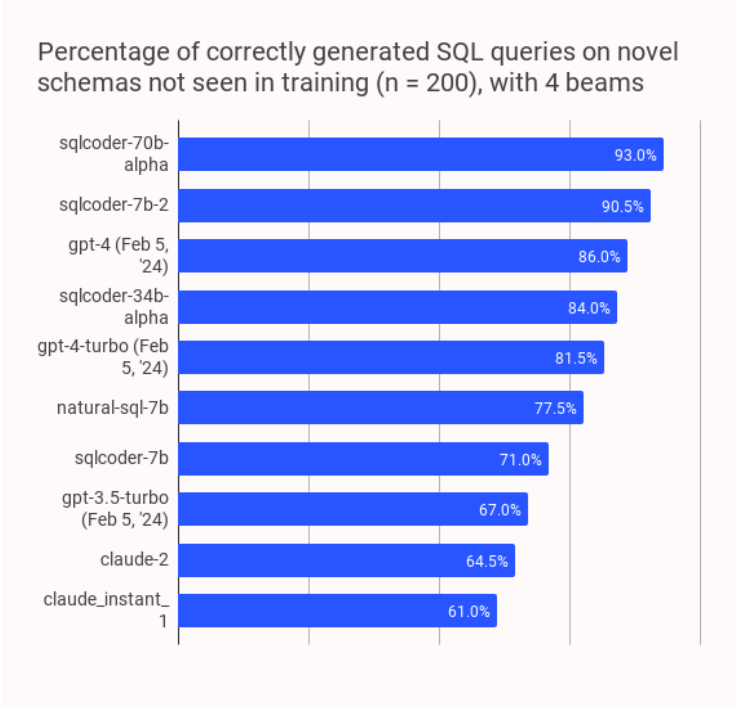
The objective of this project is to utilise a versatile Language Model (LLM) capable of generating SQL queries from natural language questions, taking into account the specific database schema provided by the user. The system will accept both natural language questions and the corresponding database schema as input. It will then translate these questions into SQL queries tailored to the provided schema, enabling users to retrieve relevant information from diverse database structures. By incorporating the database schema into its processing, the model aims to offer flexibility and adaptability, allowing it to generate SQL queries for a wide range of database schemas and domains. This project seeks to streamline the querying process by bridging the gap between natural language understanding and database interactions, facilitating efficient data retrieval and analysis across various database structures.

## Methodology :

**Base Model :** StarCoder and StarCoderBase are Large Language Models for Code (Code LLMs) trained on permissively licensed data from GitHub, including from 80+ programming languages, Git commits, GitHub issues, and Jupyter notebooks. Similar to LLaMA, we trained a ~15B parameter model for 1 trillion tokens. The StarCoderBase model was fine-tuned for 35B Python tokens, resulting in a new model that we call StarCoder.

**Instruction-tuned Model** : SQLCoder is a 15B parameter model that is fine-tuned on a base StarCoder model. It slightly outperforms gpt-3.5-turbo for natural language to SQL generation tasks on the sql-eval framework, and outperforms popular open-source models. It also significantly outperforms text-davinci-003, a model that’s more than 10 times its size.

## Relative Comparison Across Models:




## Relative Comparison Across Queries:

| query_category | gpt-4 | defog-sqlcoder | gpt-3.5-turbo | defog-easy |
|---|---|---|---|---|
| group_by | 82.9 | 77.1 | 71.4 | 62.9 |
| order_by | 71.4 | 65.7 | 60.0 | 68.6 |
| ratio | 62.9 | 57.1 | 48.6 | 40.0 |
| table_join | 74.3 | 57.1 | 60.0 | 54.3 |
| where | 80.0 | 65.7 | 62.9 | 60.0 |

## Model configuration:

Architecture : Starcoder
Quantization: 4 bits (non-sparse)
Max_context_length = 8192
Embedding Length = 6144
Number of Layers = 40
Head_count = 48
Head_count_kv = 1
Parameters = 15 Billion

## Model Preset ( ON DEVICE ) :

Context length = 2048 ( Not utilising the full context length due to LLM Hallucination)
n_gpu_layers = 10 ( Number of model layers to offload onto GPU)
GPU RAM usage : 8.0 GB to 8.5 GB
Peak CPU Load : 80%

### Prompt Format:

{System}
###
Instruction:
{User}
###
Response:
{Assistant}

**Note : Prefix and suffixes are concatenated to user_prompt according to above prompt format.**

### Prompt templates:

**System Prompt:**

You are a helpful AI assistant expert in querying SQL Databases to find answers to user's questions.
1. Create and execute a syntactically correct SQL Server query.
2. Limit the results to 10 unless specified otherwise.
3. Order the results by a relevant column to ensure the most interesting examples are returned.
4. Only request specific columns relevant to the query.
5. Not perform any Data Manipulation Language (DML) operations such as INSERT, UPDATE, DELETE, or DROP.
6. Double-check my queries before execution and provide a response based on the query results.
7. If a question doesn't relate to the database, I'll respond with "I don't know".

8. If a question is meaningless or empty, I'll respond with "Meaningless".

**User Prompt :**

### Task
Generate a SQL query to answer [QUESTION]{input_text}[/QUESTION]

### Database Schema
The query will run on a database with the following schema:
{db_schema}

### Answer
Given the database schema, here is the SQL query that
[QUESTION]{input_text}[/QUESTION]
[SQL]

**Note : Both the fields input_text and db_schema are provided through the LLM_SQL interface.**

## GuardRails:

1.) Model will not generate any queries which would affect the integrity of the database like INSERT, UPDATE and DROP queries.

2.) Model will generate some random query if the question provided by the user is meaningless or not related to the database.

## Implementation:

- We have used LM studio for loading the model and creating API for connecting to the local port where the model is currently running.
- We have built the interface using Python lightweight Framework, **Flask** which is suitable for this application as it does not need to handle large number of requests simultaneously.
- All the Generated queries are executed on postgres server of user's choice with the help of psycopg2 library.

**Chat completion with local API key :**

```
client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")

completion = client.chat.completions.create(
   model="model-identifier",
   messages=[
   {"role": "system", "content": system_prompt},
```

```
    {"role": "user", "content": prompt}
  ],
  temperature=temper,
)
```

## Functionalities :

1) Schema Learning and SQL Query Generation:

    - User provides the schema of the database to the model.
    - The model learns the features of the schema, such as table names, column names, data types, and relationships.
    - Based on the provided schema, the model generates SQL queries relevant to the user's natural language questions.

2) Connection to PostgreSQL Server:
    - Users have the option to connect to any PostgreSQL server where the database schemas and records in the tables exist.
    - The system facilitates the connection by prompting the user to provide necessary credentials and connection details.

3) Query Editing:
    - After receiving the SQL query generated by the model, the user has the option to review and edit it.
    - If the user identifies any mistakes or wishes to modify the query for specific requirements, they can make the necessary changes.
    - User would be able to provide Feedback score for the SQL query generated by the LLM.

4) Confirmation and Query Execution:
    - Once the user is satisfied with the generated or edited SQL query, they have the option to execute the query.
    - Upon clicking the "execute query" button, the query is sent to the specified PostgreSQL server for execution.

5) Query Execution and Result Display:
    - If the SQL query is correct and successfully executed on the server, the results are retrieved.
    - The results are displayed on the interface for the user to view, analyse, and interpret.

6) Error Handling and Query Adjustment:
    - If the SQL query generated by the model is incorrect or encounters an error during execution, the system captures the error message.
    - The error message is presented to the user, indicating the issue with the query.
    - The user can then edit the query based on the error message to resolve the issue and achieve the desired query outcome.
    - Based on the error, the user can think of a better prompting strategy to produce the correct SQL Query.

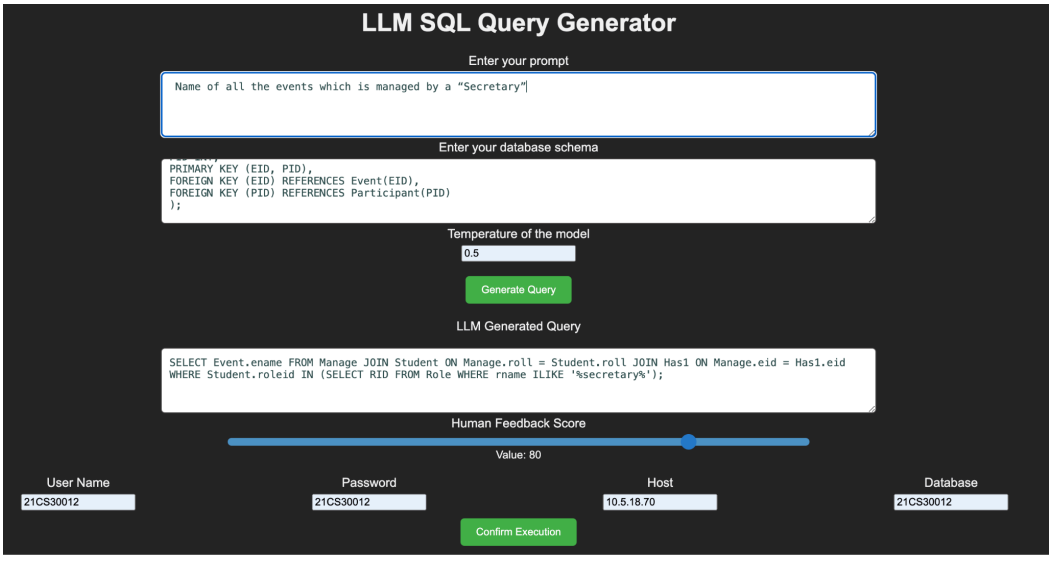
7) Temperature Parameter for Query Randomness:

- The system allows the user to specify a temperature parameter that influences the level of randomness in the SQL query generated by the language model.
- The temperature parameter controls the diversity of the generated queries, with higher temperatures leading to more varied and potentially unconventional queries, while lower temperatures tend to produce safer and more predictable queries.
- By adjusting the temperature parameter, users can fine-tune the balance between query creativity and adherence to conventional query structures, catering to their preferences and requirements for query generation.

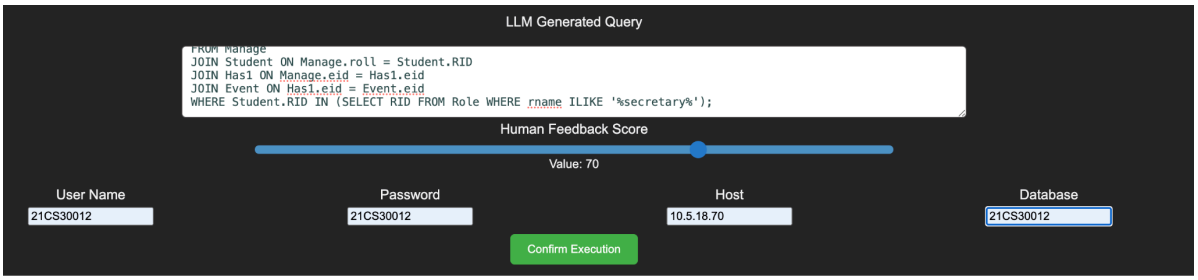
8) Storing users data for better training :

- The prompt,  generated SQL query, edited SQL Query and the feedback score are stored by application and are saved to a file.
- This information would help the organisation to identify the incorrect inferences the model is making, it helps them to redesign their training strategies.
- This data would also be useful for fine-tuning the model further and more insights into evaluating the model.

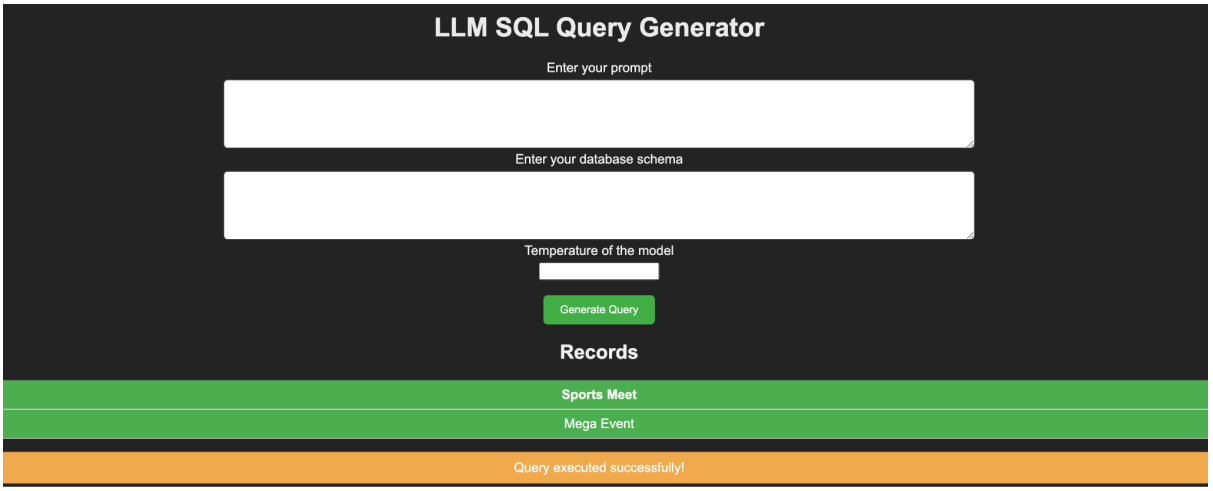
# Results/ Demonstrations :



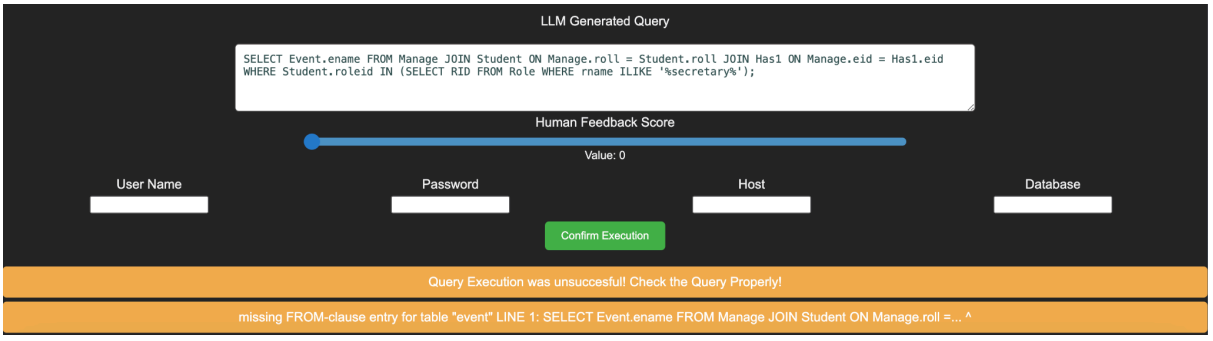

- User entered the prompt along with the database schema , SQL Query was generated by the LLM.
- Users can verify the query or can directly execute it to have error analysis and provide the feedback score.



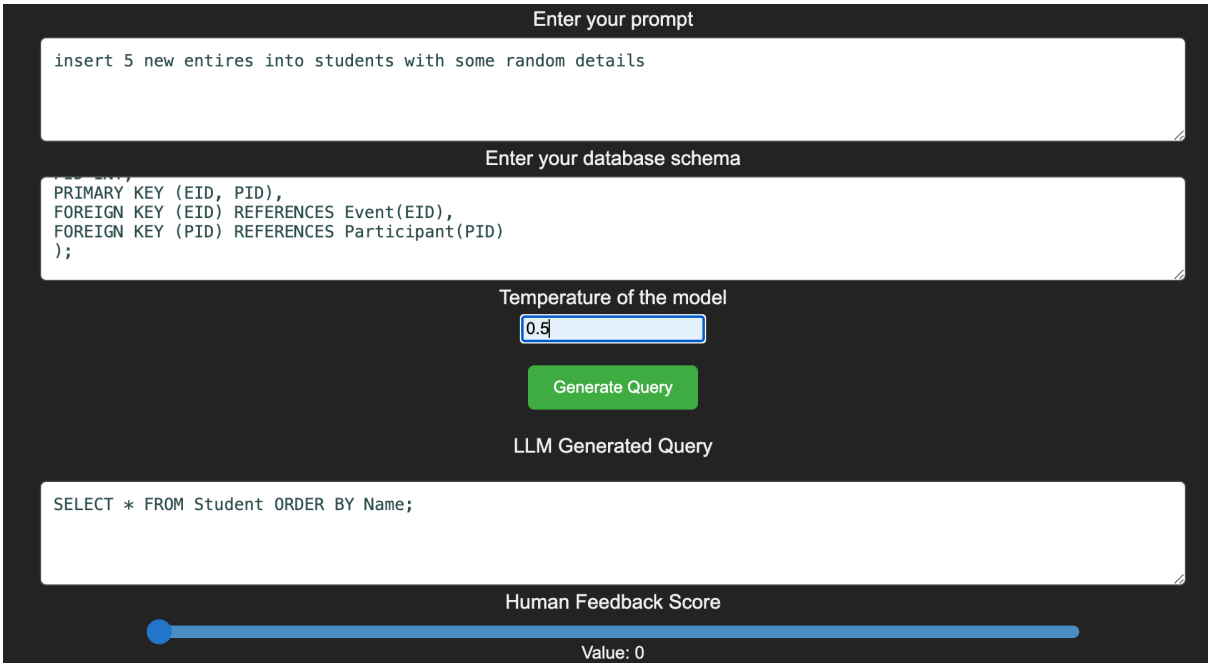

- User can edit the generated sql query if he feels it is incorrect and then enter the postgres server credentials for executing the query.



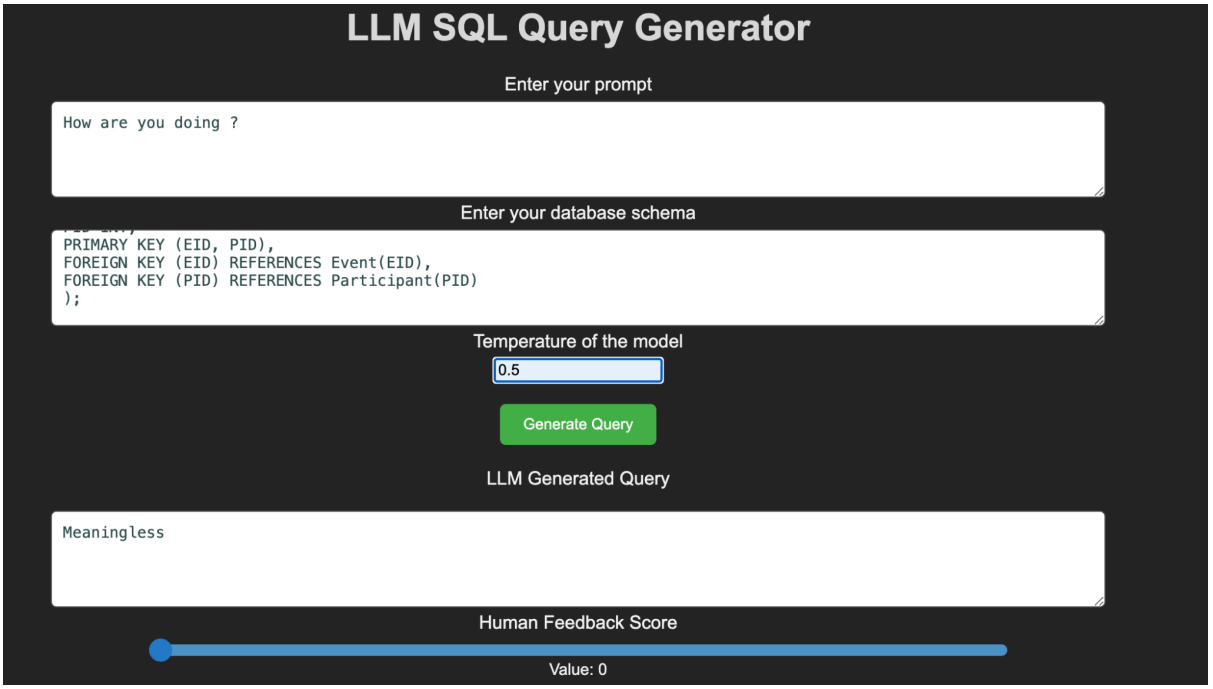

- If the execution is successful, the user would be able to see the results on the interface.



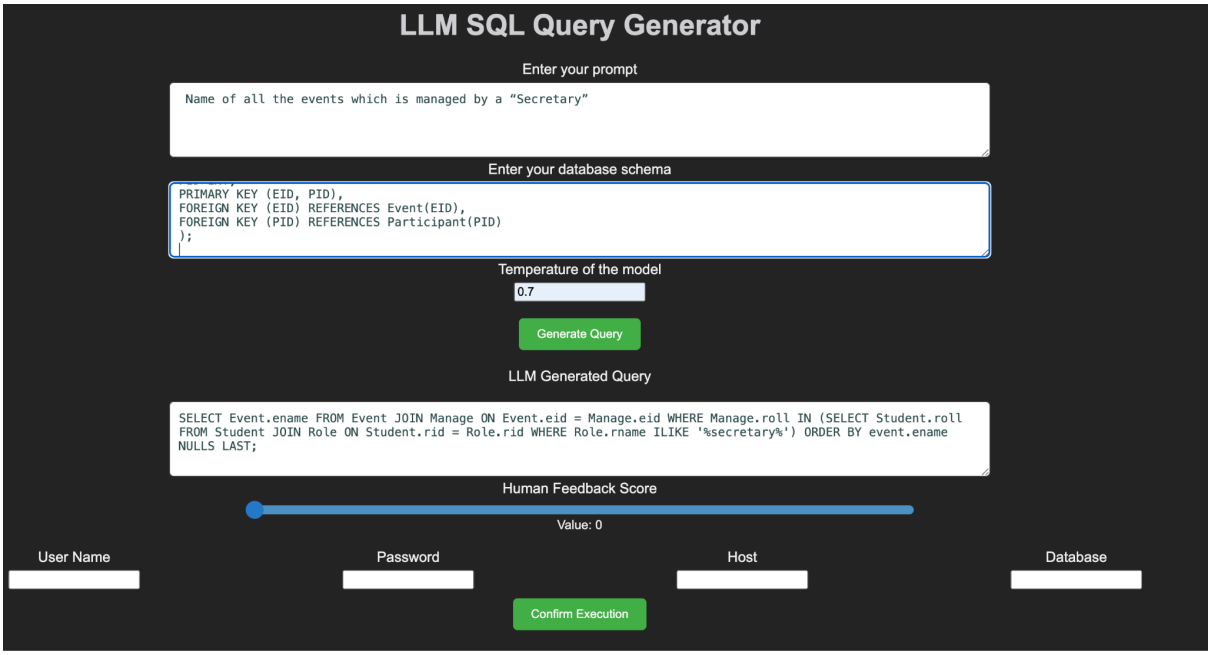

- If the execution is unsuccessful , the user would be able to see the error message on the interface, which would help him edit the query or provide a better prompt to the model.




- Model generates some random query when prompted by a question which affects the integrity of the database.




- Model responds with meaningless when prompted by an irrelevant question.




- Model generates a different query if the temperature provided by the user to the model is changed. Maintaining low temperature is preferable.

**Github Repository : [https://github.com/devichand579/LLM_SQL](https://github.com/devichand579/LLM_SQL)**

# References :

1.) Defog-ai - https://github.com/defog-ai/sqlcoder
2.) SQLCoder-2–7b: How to Reliably Query Data in Natural Language, on Consumer Hardware-https://medium.com/use-ai/sqlcoder-2-7b-how-to-reliably-query-data-in-natural-language-on-consumer-hardware-cb352a3cf3ab
3.) LM Studio - https://lmstudio.ai/docs/welcome
4.) FINE-TUNING LANGUAGE MODELS FOR CONTEXT-SPECIFIC SQL QUERY GENERATION- https://arxiv.org/pdf/2312.02251.pdf
5.) sqlcoder/hugging face - https://huggingface.co/defog/sqlcoder-7b-2