ENHANCING TEXT TO SQL CONVERSION WITH TRANSPARENT AND OPTIMIZED LLM FRAMEWORKS

PROJECT REPORT
20CDP81-Project Work II Phase II

Submitted by

MUGESHKRISHNA N
21CDR027

SANKEERTHAN S
21CDR042

TAMILARASAN U
21CDR053

in partial fulfillment of the requirements

for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND DESIGN

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE – 638 060

MARCH 2025

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE -638060
MARCH 2025

BONAFIDE CERTIFICATE

This is to certify that the Project report entitled **“ENHANCING TEXT TO SQL CONVERSION WITH TRANSPARENT AND OPTIMIZED LLM FRAMEWORKS”** is the bonafide record of project work done by **MUGESHKRISHNA N (Reg no: 21CDR027), SANKEERTHAN S (Reg no: 21CDR042) and TAMILARASAN U (Reg no: 21CDR053)** in partial fulfillment of the requirements for the award of the Degree of Bachelor of Engineering in **Computer Science and Design** of Anna University, Chennai during the year 2024 - 2025.

SUPERVISOR **HEAD OF THE DEPARTMENT**
(Signature with seal)

Date:

Submitted for the end semester viva-voce examination held on ____________________________

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE -638060
MARCH 2025

DECLARATION

We affirm that the Project report titled **"ENHANCING TEXT TO SQL CONVERSION WITH TRANSPARENT AND OPTIMIZED LLM FRAMEWORKS"** being submitted in partial fulfillment of the requirements for the award of Degree of Bachelor of Engineering is the original work carried out by us. It has not formed part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Date:

MUGESHKRISHNA N
(Reg.No: 21CDR027)

SANKEERTHAN S
(Reg.No : 21CDR042)

TAMILARASAN U
(Reg.No:21CDR053)

I certify that the declaration made by the above candidates is true to the best of my knowledge.

Date: Name & Signature of the Supervisor with seal

ACKNOWLEDGEMENT

We express our sincere thanks and gratitude to our beloved Correspondent, Thiru. **A.K. ILANGO, B.Com., M.B.A., L.L.B.,** for giving us the opportunity to pursue this course.

We are extremely thankful with no words of formal nature to the dynamic Principal, **Dr.V.BALUSAMY, MTech, Ph.D.,** for providing the necessary facilities to complete our work.

We would like to express our sincere gratitude to our respected Head of the Department of Computer Science and Design **Dr.R.THANGARAJAN B.E., M.E., Ph.D.,** for providing necessary facilities.

We also express our sincere thanks to our project coordinator, **Mrs.M.MEENALOCHINI B.E., MTech., (Ph.D)., Assistant Professor** for her constant encouragement for the development of our project.

We extend our thanks to our guide **Dr.P.JAYANTHI B.E., M.E., Ph.D., Associate Professor** for her encouragement and valuable advice that made us carry out the project work successfully. Her valuable ideas and suggestions have been very helpful in the project. We are grateful to all the faculty members of the Computer Science and Design Department, for support

ABSTRACT

Text-to-SQL conversion using NLP and ML has advanced significantly with models like GPT-4, BERT, and T5, enabling natural language queries to be transformed into SQL. However, these models often function as black boxes, making it difficult to diagnose errors or understand the underlying mechanics of semantic parsing. This lack of transparency poses a critical challenge in fields like finance and healthcare, where SQL accuracy is essential.

To address this issue, a new framework is proposed that integrates advanced large language model (LLM) techniques with interpretability-focused tools, enhancing transparency. At its core, the Heterogeneous Graph-to-Abstract Syntax Tree (HG2AST) structure converts SQL queries into Abstract Syntax Trees (ASTs). This structured representation allows the model to better comprehend the logical relationships within SQL queries, improving both performance and interpretability. In addition, the Line Graph Enhanced Encoder (LGESQL) enhances the model's ability to recognize relationships between SQL components, such as tables, joins, and conditions, further increasing accuracy.

Fine-tuning techniques like Low-Rank Adaptation (LoRA), PEFT (Parameter-Efficient Fine-Tuning), and XFormers are also utilized to enhance model performance without requiring extensive retraining. These methods allow the model to adapt to specific domains, improve computational efficiency, and handle complex, nested SQL queries more effectively.

The combination of HG2AST, LGESQL, and fine-tuning techniques results in significant improvements on benchmark datasets like Spider and WikiSQL, achieving higher accuracy and greater transparency. This approach transforms text-to-SQL models from opaque, black-box systems into more interpretable and explainable AI solutions, enabling developers to better understand and optimize them for diverse domains and database structures

INDEX

LIST OF FIGURES

LIST OF ABBREVIATIONS

LLM	-	Large Language Model
HG2AST	-	Heterogeneous Graph-to-Abstract Syntax Tree
LGESQL	-	Line Graph Enhanced Encoder SQL
LoRA	-	Low-Rank Adaptation
PEFT	-	Parameter-Efficient Fine-Tuning
MAVIDSQL	-	Model Agnostic Visualization for Interpretation and Diagnosis
AST	-	Abstract Syntax Tree
BERT	-	Bidirectional Encoder Representation from Transformers

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The integration of Natural Language Processing (NLP) and Machine Learning (ML) has significantly transformed text-to-SQL systems, making it possible for users to generate SQL queries using natural language. This advancement reduces the need for extensive SQL knowledge, enabling users across various industries to interact seamlessly with databases. Despite these improvements, conventional Large Language Models (LLMs) present a major challenge: their lack of interpretability. These models often function as black-box systems, meaning developers struggle to understand their decision-making processes. This limitation complicates debugging, fine-tuning, and ensuring the accuracy of SQL queries, which is particularly concerning in fields such as finance and healthcare, where database precision is critical.

To overcome these challenges, this project introduces an enhanced text-to-SQL framework that emphasizes both accuracy and transparency. The foundation of this system is Microsoft Phi-4K Instruct, a powerful Transformer-based model. This base model is further optimized through the application of advanced fine-tuning techniques, including X-Formers, Low-Rank Adaptation (LoRA), and Parameter-Efficient Fine-Tuning (PEFT). These methods improve computational efficiency while maintaining high accuracy, ensuring that the model is adaptable to various database structures. Additionally, the project integrates Transformer Reinforcement Learning (TRL) to combine the strengths of these fine-tuning techniques into a unified system, enhancing both performance and scalability.

A central innovation in this framework is the implementation of the Heterogeneous Graph-to-Abstract Syntax Tree (HG2AST) architecture. Traditional text-to-SQL models rely heavily on token-based representations, which can struggle to capture the structural complexities of SQL queries. In contrast, the HG2AST approach transforms SQL queries into Abstract Syntax Trees (ASTs), providing a structured and interpretable representation. This transformation allows the model to understand the relationships between different SQL components, improving query formation and making the decision-making process more transparent for developers. To further strengthen the system's ability to handle complex SQL queries, the project integrates the Line

Graph Enhanced Encoder (LGESQL). This module refines the model's understanding of database relationships, ensuring that SQL queries correctly map to table joins, conditions, and nested subqueries. By structuring the SQL query components more effectively, LGESQL enhances the accuracy and consistency of query generation, particularly for databases with intricate schemas.

Another major enhancement in this project is the integration of OpenAI's LLM capabilities with the fine-tuned Transformer model. This integration allows for prompt-based query generation, making the system more adaptable to various user inputs. Instead of relying solely on predefined patterns, the model dynamically interprets natural language prompts and generates SQL queries in real time. This feature significantly improves user interaction, making it easier for non-technical users to retrieve information from databases without requiring in-depth SQL expertise.

The training process of the model incorporates multiple optimization techniques to enhance both performance and accuracy. BERT, HG2AST, and LGESQL are employed for semantic understanding and query structuring, while prompt training using real-world database examples ensures that the model generalizes well across different datasets. Additionally, improvements in tokenization strategies help refine query formation, reducing syntax errors and improving execution accuracy. The dataset B-MC2/SQL-Create-Context from Hugging Face is used for pretraining and validation, ensuring robustness in diverse scenarios. By integrating these cutting-edge advancements, this project delivers a highly transparent and interpretable text-to-SQL model. Unlike traditional black-box models, this system allows developers to trace query generation logic, making debugging and fine-tuning significantly easier. Moreover, the enhanced architecture ensures better accuracy, making the model a practical solution for real-world applications.

This framework bridges the gap between natural language understanding and structured databases, enabling more intuitive database interactions. Whether in business intelligence, financial analytics, or healthcare data management, this system provides an accessible, scalable, and efficient solution for translating human language into precise SQL commands. Future work will focus on expanding the model's generalization capabilities through transfer learning and domain adaptation, optimizing computational efficiency for real-time applications, and incorporating multimodal inputs for even greater flexibility.

CHAPTER 2

PROJECT DESCRIPTION

The evolution of text-to-SQL conversion has significantly enhanced how users interact with databases, allowing them to generate SQL queries using natural language. However, traditional Large Language Models (LLMs) often function as black-box systems, making it difficult to understand their decision-making process. This lack of transparency poses challenges in debugging, fine-tuning, and ensuring the reliability of generated SQL queries, particularly in critical domains such as finance and healthcare. To address these issues, this project presents an advanced text-to-SQL framework built on Microsoft Phi-4K Instruct as the base Transformer model. By integrating fine-tuning techniques such as X-Formers, Parameter-Efficient Fine-Tuning (PEFT), and Low-Rank Adaptation (LoRA), the model achieves greater efficiency and adaptability. These techniques are combined through Transformer Reinforcement Learning (TRL) injection, ensuring that the strengths of these optimizations work cohesively to enhance query accuracy and efficiency.

A key innovation in this framework is the introduction of the Heterogeneous Graph-to-Abstract Syntax Tree (HG2AST) architecture, which improves the interpretability of generated SQL queries by transforming them into structured representations. This structural transformation allows the model to better understand relationships within SQL queries, ensuring logical coherence and reducing errors in complex queries involving joins, conditions, and nested structures. In addition to HG2AST, the framework employs the Line Graph Enhanced SQL Encoder (LGESQL), which further enhances the model's ability to process SQL queries accurately. LGESQL improves the model's capacity to handle structural dependencies within a query, helping it generate well-formed and semantically correct SQL statements. These enhancements contribute to the overall reliability and interpretability of the system, addressing the challenges posed by traditional black-box LLM approaches.

To further optimize query generation, the model undergoes extensive training using BERT, HG2AST, and LGESQL, along with prompt training using real-world database examples. By fine-tuning with contextualized datasets, the model improves its ability to process diverse natural language queries and generate accurate SQL statements. Additionally, OpenAI's LLM integration with the Microsoft Phi-4K Instruct model plays a crucial role in refining SQL query generation. When a user inputs a natural language prompt, the system processes it, generates an SQL query, and applies tokenization and optimization techniques to improve accuracy and execution speed. Tokenization optimization ensures that query structures are preserved while minimizing errors in syntax and logic, ultimately enhancing the reliability of query outputs.

This framework represents a significant advancement in the field of text-to-SQL conversion, offering a transparent, optimized, and scalable solution for database interaction. Unlike conventional models, which often struggle with interpretability and efficiency, this system provides a structured approach to query generation, making it easier to debug and fine-tune. The integration of advanced fine-tuning techniques, structural encoding models, and OpenAI's LLM ensures that the model not only generates highly accurate SQL queries but also provides transparency in its decision-making process. By bridging the gap between natural language and structured databases, this project contributes to making database interaction more accessible, interpretable, and adaptable across various domains. The improvements introduced in this framework set the foundation for future advancements in NLP-driven database management, offering a robust solution that meets the growing demands for accurate and efficient text-to-SQL conversion.

CHAPTER 3

LITERATURE REVIEW

Jingwei Tang et al. introduce MAVIDSQL, a visual analytics system aimed at enhancing the interpretation and diagnosis of text-to-SQL (T2S) models. T2S tasks convert natural language queries into structured SQL statements, but interpreting the model's reasoning and identifying errors is often challenging. MAVIDSQL addresses these issues by providing a comprehensive set of four visualization components that allow users to explore model predictions from multiple angles.

The first component, the Projection View, displays 2D projections of input sentences based on their semantic similarity. This view helps uncover relationships between different sentence components, offering insights into how the model interprets natural language inputs. By extracting latent semantic information, users can better understand the connections that influence the model's predictions.

The second component, the SQL Comparison View, offers a side-by-side analysis of the SQL generated by the model compared to the ground truth SQL. This view highlights syntactic and structural differences between the predicted and actual SQL statements, making it easier to pinpoint areas where the model's output deviates from the expected query.

The Model Performance Statistics View serves as the third component, presenting evaluation metrics like Execution-Accuracy (ESM) to give users a quantitative perspective on the model's performance. This view helps identify trends in model accuracy across different queries, allowing users to detect patterns of underperformance.

Lastly, the Raw Data View provides access to the raw input data and model predictions, enabling detailed exploration of specific error patterns. This component allows users to drill down into the model's outputs to identify particular areas of misinterpretation or error.

While MAVIDSQL effectively aids in diagnosing and understanding T2S models, its reliance on complex visualizations may present a learning curve for users unfamiliar with advanced visualization tools. Future improvements could focus on making these tools more accessible to a broader audience, streamlining the interpretability process without compromising the system's diagnostic power.

Ruisheng Cao et al. introduce the HG2AST Framework, designed to improve text-to-SQL conversion models, particularly in complex multi-table scenarios. Text-to-SQL tasks often struggle with capturing relational dependencies between tables, but HG2AST tackles this issue through two key innovations: the Line Graph Enhanced SQL (LGESQL) encoder and the Abstract Syntax Tree (AST) decoder. These components are designed to enhance the model's ability to generate grammatically correct SQL queries while effectively handling complex SQL structures.

The LGESQL encoder utilizes a line graph to represent the topological connections and contextual semantics between edges in the input graph. This technique captures both 1-hop and multi-hop relationships between tables, which is crucial for accurately generating SQL queries that span multiple tables. By distinguishing these relational dependencies, the encoder improves the model's ability to create SQL sketches that reflect the underlying database structure, thereby increasing the accuracy of SQL generation in multi-table scenarios.

The AST decoder, alongside the Grammar Tree Language (GTL) algorithm, dynamically controls the expansion of AST nodes during query generation. This flexibility allows the model to adapt its decoding process and avoid biases imposed by fixed grammatical rules. By controlling the node expansion order, the model explores multiple potential SQL paths, leading to more accurate and interpretable SQL generation. This adaptability is particularly effective in reducing errors related to SQL grammar and in handling diverse SQL structures.

In addition to the core encoder and decoder, the framework incorporates auxiliary modules that further enhance the system's understanding of SQL grammar and reduce unnecessary biases. These refinements contribute to the model's top-ranked performance across eight prominent text-to-SQL benchmark datasets, demonstrating its robustness in handling a variety of query types and database structures.

The authors propose several potential future improvements to the framework. One suggestion involves incorporating higher-order structural knowledge of edges into the heterogeneous inputs to further boost pre-trained language models' performance. Another area for improvement is leveraging graph structure knowledge of semantic dependencies in natural language questions to refine graph encoding. The AST decoder’s node selection process could also be enhanced by adopting more strategic, rather than random, sampling techniques during query generation.

Despite its success, the framework still faces challenges in predicting executable SQL queries that involve specific values. To address this, techniques like entity linking and value normalization are proposed to manage variations in entity representation, such as morphological changes, synonyms, and implicit mentions. Overall, the HG2AST framework represents a significant advancement in the development of interpretable and high-performing text-to-SQL models, particularly in scenarios involving multi-table queries and resource constraints.

Manasi Ganti et al., in their study "Evaluating Text-to-SQL Model Failures on Real-World Data," examine the substantial performance gaps between text-to-SQL models in real-world applications versus academic benchmarks like Spider. While models such as GPT-4 are effective at translating natural language queries into SQL, they experience significant accuracy declines when applied to real-world datasets, which typically involve more complex schemas and longer contexts. The authors attribute these challenges to factors like extended schema lengths, unclear question formulations, and increased query complexity, which are not fully represented in existing academic benchmarks.

Through their analysis of model failures on customer logs, Ganti et al. discovered that text-to-SQL models perform on average 30% worse on real-world data compared to Spider, exposing the limitations of current benchmarks. They identify three key challenges that hinder model performance in practical applications: longer context lengths, ambiguous questions, and more complex query structures. These elements are less prevalent in academic benchmarks, suggesting a need for more representative testing frameworks.

To address these shortcomings, the authors developed a new benchmark based on manually labeled customer logs. This dataset includes 50 SQL queries—20 non-join and 30 join queries—each with three variations in phrasing, resulting in a total of 200 queries. Schema sizes range from 5 to over 300 columns, testing the models' abilities to manage increasingly complex databases, thus better reflecting real-world conditions.

Ganti et al. evaluated the performance of several prominent models, including GPT-4, GPT-3.5, Starcoder by BigCode, and NSQL Llama-2, using both the new benchmark and the Spider dataset. The results show a consistent decrease in execution accuracy as schema size grows, with a drop of approximately 0.5 percentage points for every additional 10 columns. This demonstrates that current models struggle with the large tables and schema complexities common in real-world applications.

In conclusion, the study highlights the limitations of existing text-to-SQL models and benchmarks when applied to real-world data, underscoring the need for more rigorous evaluation frameworks that accurately reflect practical challenges. The new benchmark proposed by Ganti et al. aims to fill this gap and inspire future improvements in text-to-SQL model performance.

CHAPTER 4

REQUIREMENTS SPECIFICATION

4.1 HARDWARE REQUIREMENTS

Processor Type	:	Intel i5
Speed	:	3.40GHZ
RAM	:	16GB DD4 RAM
Hard disk	:	500 GB

4.2 SOFTWARE REQUIREMENTS

Operating System	:	Windows 10
Front end	:	Google Colab Notebook
Coding Language	:	Python

4.3 SOFTWARE DESCRIPTION

4.3.1 GOOGLE COLAB NOTEBOOK

The Google Colab Notebook is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations and narrative text. Uses include data cleaning and transformation, data visualization, machine learning, and much more.

The software requirement specification is created at the end of the analysis task. The function and performance allocated to software as part of system engineering are developed by establishing a complete information report as functional representation, a representation of system behavior, an indication of performance requirements and design constraints, and appropriate validation criteria.

4.3.2 FEATURES OF GOOGLE COLAB NOTEBOOK

- In-browser editing for code, with automatic syntax highlighting, indentation, and introspection
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc.

4.3.3 NOTEBOOK DOCUMENTS

Notebook documents contain inputs and outputs of an interactive session and additional text with the code that is not meant for execution. Notebook files can serve as a complete computational record, having executable code, explanatory text, mathematics, and rich representations of objects. The documents are JSON files saved with the ipynb extension. JSON is a plain text format so they can be version-controlled and shared with colleagues.

4.3.4 PYTHON DEFINITION

Python is a high-level programming language designed to be easy to read and simple to implement. It is open source, which means it is free to use, even for commercial applications. Python can run on Mac, Windows, and Unix systems and also ported to Java and .NET.

Python is considered a scripting language, like Ruby or Perl and is often used for creating Web applications and dynamic Web content. It is also supported by a number of 2D and 3D imaging programs, enabling users to create custom plug-ins and extensions with Python. Scripts written in Python (.PY files) can be parsed and run immediately. They can also be saved as compiled programs (.PYC files), which are often used as programming modules that can be referenced by other Python programs.

PYTHON FEATURES

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. To have listed below a few essential features.

1) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program can be simply typed as print("Hello World"). It will take only one line to execute, while Java or C takes multiple lines.

2) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

3) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

4) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards making new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

5) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object- oriented procedure helps programmers to write reusable code and develop applications in less code.

6) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into bytecode, and any platform can use that byte code.

7) Large Standard Library

It provides a vast rang–e of libraries for various fields such as machine learning, web development, and also for scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

ANACONDA

Anaconda Cloud is a package management service by Anaconda. Cloud makes it easy to find, access, store and share public notebooks, environments, and anaconda and PyPI packages. Cloud also makes it easy to stay current with updates made to the packages and environments that are being used. Cloud hosts hundreds of useful Python packages, notebooks, projects and environments for a wide variety of applications.

It is possible to build new Conda packages using Conda-Build and then upload the packages to the cloud for quick sharing with others or for personal access from anywhere. The Anaconda Cloud command line interface (CLI), known as anaconda-client, enables the management of an Anaconda Cloud account. It includes tasks such as authentication, token generation, package uploading, downloading, removal, and search. Users can connect to and manage their Anaconda Cloud accounts, upload created packages, and generate access tokens to grant access to private packages

CHAPTER 5

DATASETS

5.1 DESCRIPTION OF DATASET:

This dataset builds on WikiSQL and Spider, offering 78,577 examples of natural language queries, SQL CREATE TABLE statements, and SQL queries that answer the questions using the CREATE TABLE statements as context. It was designed for text-to-SQL LLMs to reduce hallucinations of column and table names, a common issue when trained on standard text-to-SQL datasets. By providing only the CREATE TABLE statement, which includes table names, column names, and data types, the dataset aims to ground models effectively without exposing actual data rows, thus minimizing token usage and protecting sensitive information.

Cleansing and augmentation were performed by using SQLGlot to parse queries from WikiSQL and Spider into tables and columns. Data types for columns were inferred based on usage in the queries (e.g., with operators like >, < or functions like MIN(), MAX(), SUM()). Where no types were inferred, VARCHAR was assigned as the default. SQLGlot was used again to ensure queries and CREATE TABLE statements parsed correctly.

In cases where column names were missing (e.g., SELECT *), a default Id column was added. Generic table names like "table" were replaced with variations like "table_name_1" for clarity.

Planned improvements include augmenting data with different SQL dialects, parsing more complex data types, and cleaning up columns with numerical names. Dialects can be added via SQLGlot to diversify the dataset. Additional informative contexts beyond CREATE TABLE statements are also a future goal.

5.2 ATTRIBUTES OF THE DATASET :

The **b-mc2/sql-create-context** dataset includes a range of synthetic tables that replicate real-world database structures. These tables enable a variety of SQL queries focused on user profiles, social media activity, follower relationships, and course management. The key attributes are organized as follows:

User Profiles Table:

- **name**: Full name of the user (VARCHAR).
- **email**: User's email address (VARCHAR).
- **followers**: Number of followers a user has (INTEGER).
- **uid**: Unique identifier for the user (VARCHAR).

Follows Table:

- **f1**: Unique identifier of the follower (VARCHAR).
- **f2**: Unique identifier of the user being followed (VARCHAR).

Tweets Table:

- **uid**: User's unique identifier for tracking their tweets (VARCHAR).
- **text**: The content of the tweet (VARCHAR).
- **createdate**: Date and time the tweet was posted (VARCHAR).

Course-related Tables:

- **student_id**: Unique identifier for the student (VARCHAR).
- **course_id**: Unique course identifier (VARCHAR).
- **registration_date**: Date when the student registered for the course (VARCHAR).
- **course_name**: Name of the course (VARCHAR).

These attributes facilitate complex SQL queries for tasks such as social media data analysis, follower relationships, tweet tracking, and academic course registrations.

CHAPTER 6

EXISTING SYSTEM

6.1 Heterogeneous Graph Encoder

A heterogeneous graph encoder (HGE) is a powerful tool in text-to-SQL systems, where natural language queries are transformed into SQL queries that interact with a database. Text-to-SQL processing requires an understanding of both the user's intent expressed through natural language and the structure of the database, including tables, columns, and relationships. The challenge is converting human language into SQL statements that correctly map to the database schema. A graph-based approach is effective here, as it allows the various elements involved in the query—such as words and database schema components—to be represented as nodes, while the relationships between them, such as table-column links or semantic connections between words, are represented as edges. For example, in a query like "Show me all students enrolled in Computer Science courses," tables like "Students" and "Courses" and their joining column "course_id" must be identified.

In practice, these graphs are heterogeneous, meaning they contain different types of nodes (query words, database tables, columns) and edges (table-to-column, word-to-column). This is where the heterogeneous graph encoder becomes essential. The HGE is designed to handle diverse node and edge types, learning rich representations that capture the relationships between them. The process starts by representing both the natural language query and the database schema as a graph. Each word in the query and each element of the schema becomes a node, and the relationships between these nodes are represented as edges. The HGE processes the graph, generating embeddings for each node and edge that encode their contextual meaning and the relationships between them. By encoding the structure of the query and database, the HGE helps the system understand how the query relates to specific tables or columns. This enables the model to generate the correct SQL query, identifying the required tables, columns, conditions, and joins. The use of HGEs in text-to-SQL models allows them to handle complex queries involving multiple joins, conditions, or even nested SQL structures. HGEs capture the semantic nuances of the query as well as the structural relationships in the database, making the system more accurate and adaptable. Their ability to manage different types of relationships also means HGEs generalize well across different databases, improving the flexibility of text-to-SQL systems. This makes heterogeneous graph encoders a key technology in enabling more natural interactions between users and databases, leading to better data retrieval.

6.2 Golden Tree-Oriented Learning

The Golden Tree-oriented Learning (GTL) algorithm is an advanced approach for translating natural language text into SQL queries. This algorithm uses a tree-based structure, where each node represents a component of the SQL query, and the tree is built step-by-step following predefined grammar rules. At the heart of GTL is the process of expanding and traversing this tree, which is managed using Depth-First Search (DFS). The DFS approach is particularly suited for this task because SQL queries have a hierarchical, top-down structure similar to how natural language sentences are constructed.

The first step in GTL involves identifying the frontier node set, which consists of unexpanded nodes that are available for further processing at each timestep. The algorithm starts with a single root node that typically represents the entire SQL query. As each node is expanded according to the grammar rules, its child nodes are added to the frontier node set, ready to be processed in the next step. If a node has no more unexpanded children or reaches a terminal state, the algorithm backtracks to its parent node, continuing this process until the entire SQL query tree is constructed. This method ensures that all parts of the SQL query are handled in an efficient, top-down manner, much like how grammatical structures are formed in natural language.

Once the frontier node set is identified, GTL introduces flexibility in how the next node is selected for expansion. Traditional approaches often follow a rigid, predetermined order based on grammar rules, where, for example, the select clause is always expanded before the from clause in SQL queries. However, GTL deviates from this convention by randomly selecting a node from the frontier set during training. This randomness allows the algorithm to explore different ways of constructing SQL queries and makes it more adaptive to various query structures. By incorporating this flexibility, GTL can generalize better across different types of queries, improving its ability to accurately translate natural language into SQL queries.

6.3 Advancing Text-to-SQL: Enhancing Financial Query Automation with Large Language Models

Text-to-SQL (Text2SQL) is a critical technology in the financial sector, enabling the translation of human language into structured SQL queries to extract precise data from complex databases. Large language models (LLMs), such as GPT and Codex, have greatly advanced this field by improving contextual understanding and code generation. This is especially important in finance, where queries can be ambiguous, involve intricate SQL structures, or use specialized vocabulary. LLMs outperform traditional rule-based systems in handling these complexities, offering more accurate and reliable SQL generation.

Despite these advancements, three major challenges remain: performance benchmarking, evaluation methodologies, and input optimization. Our research introduces two novel metrics to evaluate SQL query similarity, focusing on the semantic equivalence of queries rather than exact matches. This approach is particularly valuable in finance, where different SQL structures can yield the same results. By analyzing LLM performance on standard datasets like Spider, WikiSQL, and a specialized financial sub-dataset, This highlight the limitations of current models, especially in handling domain-specific queries and complex financial databases.

Finally, this explores prompt engineering to optimize how queries are presented to LLMs, showing that small adjustments in phrasing can lead to significant improvements in SQL generation. This insight suggests that business users, with minimal training, can enhance their interaction with financial databases. Overall, research paves the way for future advancements in financial query automation, as LLMs continue to evolve and become more integrated into Text-2-SQL systems.

CHAPTER 7

PROPOSED SYSTEM

7.1 PROPOSED MODEL ARCHITECTURE:

The proposed model architecture for enhancing text-to-SQL conversion with transparent and optimized LLM frameworks is designed to address the challenges of accuracy, efficiency, and interpretability in SQL query generation. The architecture is built on Microsoft Phi-4K Instruct as the base Transformer model, integrating advanced fine-tuning techniques and structural encoding methods to improve query generation. The model leverages X-Formers, Parameter-Efficient Fine-Tuning (PEFT), and Low-Rank Adaptation (LoRA) techniques, which are injected using Transformer Reinforcement Learning (TRL) to optimize performance while reducing computational overhead. These fine-tuning approaches ensure efficient adaptation to diverse datasets, enhancing the model's ability to generate precise SQL queries from natural language inputs.

At the core of the architecture is a multi-stage processing pipeline that begins with natural language input from the user. When a query is entered, OpenAI's LLM integration processes the prompt to understand the user's intent and structure the input into a form suitable for SQL query generation. This initial processing stage ensures that contextual nuances in the natural language prompt are effectively captured, improving the quality of query generation. The processed input is then passed through the base Transformer model, where the TRL-injected fine-tuning techniques refine the model's response by leveraging domain-specific training and learned optimizations.

The architecture incorporates the Heterogeneous Graph-to-Abstract Syntax Tree (HG2AST) module, which plays a crucial role in enhancing interpretability. This module converts generated SQL queries into structured representations, providing a clear understanding of query components and their relationships. By representing queries as Abstract Syntax Trees (ASTs), the model improves its ability to handle complex SQL structures, including nested queries, joins, and conditional statements. The HG2AST transformation enables better debugging and fine-tuning, ensuring that query outputs align with expected database structures.

To further refine query generation, the model integrates the Line Graph Enhanced SQL Encoder (LGESQL), which improves structural coherence and accuracy. LGESQL enhances the model's ability to recognize relationships between database entities, ensuring that SQL queries maintain semantic and syntactic integrity. This encoding method helps the model effectively process query dependencies, reducing the likelihood of generating incorrect or inefficient SQL statements. By combining HG2AST and LGESQL, the proposed architecture ensures that query outputs are both logically sound and transparent, making it easier to interpret and optimize results.

Model training is conducted using a combination of BERT, HG2AST, and LGESQL, along with prompt training on real-world database examples. This training approach ensures that the model generalizes well across different database schemas, enabling it to adapt to various query structures and user inputs. Tokenization and optimization techniques are applied at multiple processing stages to enhance query accuracy and execution efficiency. By improving tokenization strategies, the model minimizes syntax errors and ensures that generated queries are well-formed and executable across different database environments.

Overall, the proposed architecture as shown in fig 7.1 provides a structured and transparent approach to text-to-SQL conversion, addressing key challenges in accuracy, efficiency, and interpretability. The integration of advanced fine-tuning techniques, structural encoding modules, and OpenAI's LLM ensures that the model delivers high-quality SQL queries while providing developers with the tools needed to understand and optimize query generation. This framework represents a significant advancement in NLP-driven database interaction, offering a scalable and adaptable solution for various industries requiring efficient and reliable text-to-SQL conversion.

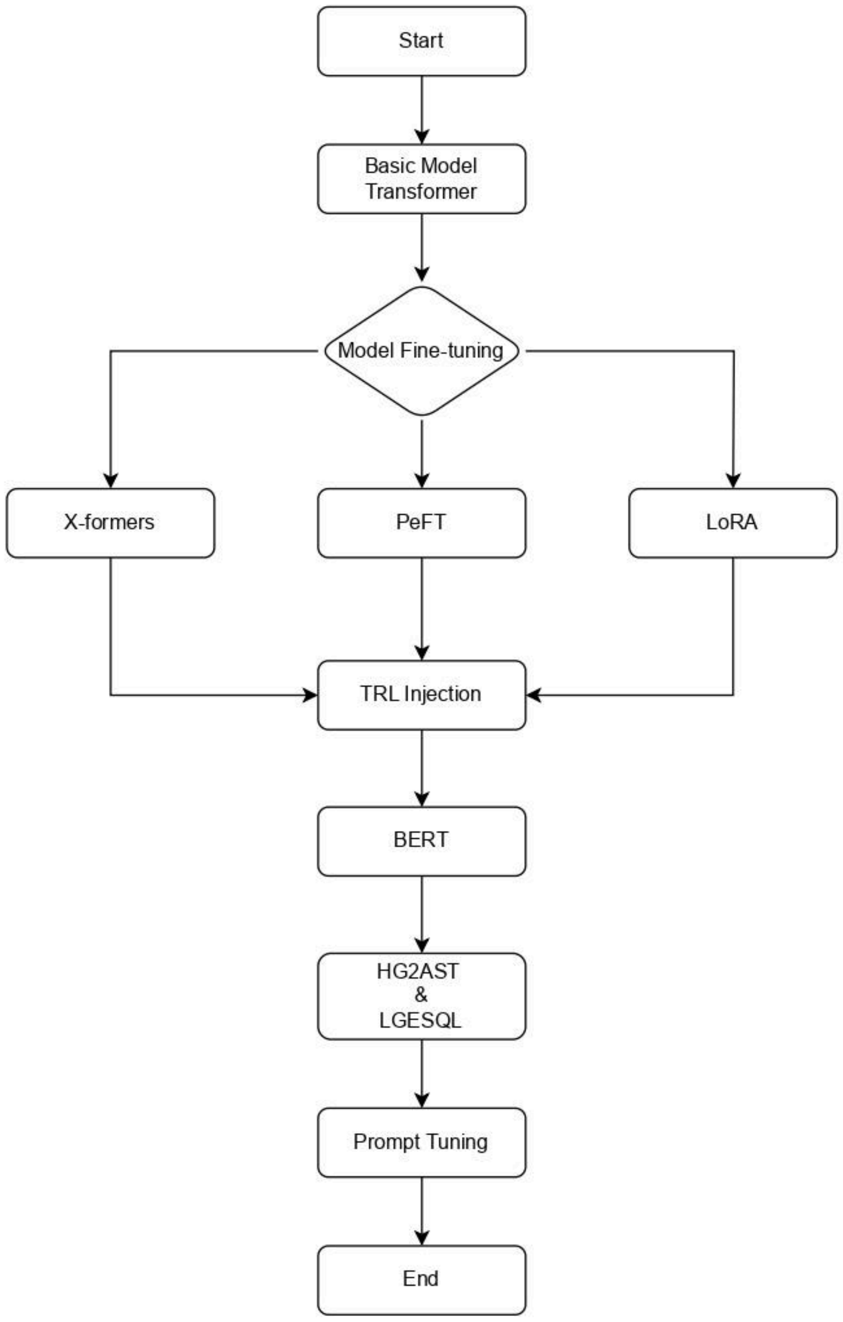


Fig 7.1 - Proposed Model Architecture

7.2 X-Formers Integration :

Integrating XFormers into natural language processing (NLP) models for Text-to-SQL query generation introduces a significant advancement in memory efficiency without compromising accuracy. Text-to-SQL, which involves converting natural language into structured SQL queries, often requires complex transformer models like GPT, BERT, or T5 that are typically resource-intensive. The challenge with these large language models (LLMs) is their high memory consumption, particularly when scaling them for real-time query generation tasks involving large datasets. XFormers, a set of transformer layers optimized for memory usage, addresses this challenge by significantly reducing the memory footprint while maintaining the high accuracy required for effective SQL query generation. This integration allows transformer models to handle larger batches, longer sequences, and more intricate queries within the same hardware constraints, which is particularly crucial in resource-limited environments or large-scale database systems. By improving memory efficiency, XFormers enable models to process more extensive and more complex natural language queries, which are often seen in financial, healthcare, or e-commerce domains, where SQL queries involve numerous joins, subqueries, or nested operations. The use of XFormers doesn't just improve scalability—it also ensures that Text-to-SQL models can operate in real-time settings, delivering faster query responses while handling dynamic and sophisticated natural language inputs. With these improvements, users, including non-technical stakeholders, can seamlessly interact with relational databases using everyday language, benefiting from quicker and more efficient query generation.

Additionally, XFormers allow for more flexibility in the fine-tuning of LLMs across various Text-to-SQL tasks, enabling better domain-specific adaptations without excessive computational strain. For instance, in financial services, where accurate and timely retrieval of transactional data or customer insights is essential, XFormers make it possible to deploy robust NLP models capable of generating SQL queries on-the-fly, even for intricate data requests, without sacrificing processing speed or accuracy.

As the demand for natural language interfaces to query databases continues to grow, especially in business intelligence, As shown in(fig 7.2) X Formers integration will play a crucial role in enhancing the overall performance and accessibility of Text-to-SQL systems.

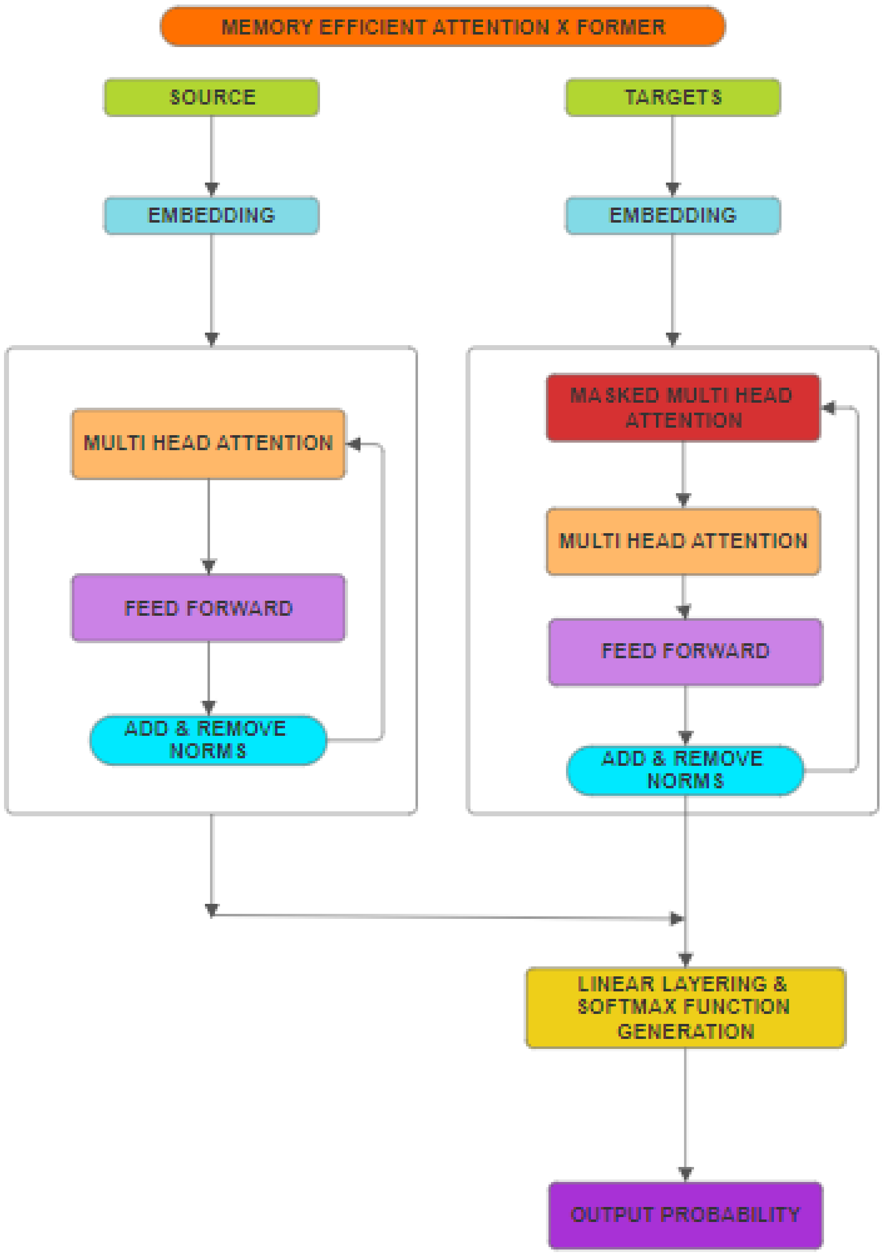


Fig 7.2 X Formers Architecture

PSEUDOCODE:

Input: Natural language query (NLQ) provided by the user.

Output: Optimized token representations with enhanced memory efficiency for downstream Text-to-SQL processing.

```
1. Preprocess input:
   - Tokenize the natural language query (NLQ) and convert to embeddings.
2. Initialize model:
   - Use a pre-trained language model (e.g., GPT/BERT).
   - Freeze its core weights.
3. Set up X-Formers:
   - Replace standard transformer layers with X-Former layers.
   - Optimize for memory and speed (e.g., sparse attention, block-wise operations).
4. Forward pass:
   - Use X-Former layers for efficient attention computation.
   - Process NLQ embeddings through the model.
5. SQL Query Generation:
   - Use a decoder to generate the SQL query from the model's output embeddings.
6. Fine-tune:
   - Train with NLQ-SQL pairs, updating only the X-Former-specific parameters.
7. Inference:
   - Given a new NLQ, pass it through the X-Former model and output the SQL query.
```

7.3 PEFT Integration for Enhanced Text-to-SQL Query Generation

Incorporating Parameter-Efficient Fine-Tuning (PEFT) into natural language processing models for Text-to-SQL tasks has revolutionized the adaptability and efficiency of these models, particularly in specialized domains. Text-to-SQL involves the automatic conversion of natural language queries into SQL statements, enabling users to interact with databases using conversational language. Traditional approaches to fine-tuning large language models (LLMs) for such tasks typically require adjusting millions or even billions of parameters, which is computationally expensive and resource-intensive. PEFT offers a solution by allowing models to be fine-tuned with far fewer parameters, enhancing their performance while significantly reducing the computational load. This method is especially beneficial for domain-specific applications where the model must adapt to complex query structures, such as in finance, healthcare, or logistics, without requiring massive datasets or retraining the entire model.PEFT works by selectively fine-tuning specific layers or modules of the pre-trained model, targeting only the parameters relevant to the SQL generation task. This selective adjustment reduces the number of parameters that need to be modified, thereby lowering memory and processing requirements. Despite fine-tuning a smaller subset of parameters, the model retains its general language understanding capabilities while becoming highly specialized in translating natural language into SQL queries. This results in a significant boost in task-specific performance, as the model can adapt quickly to new data or queries without the overhead of full-model retraining.

For Text-to-SQL tasks, PEFT enhances the model's ability to generate accurate and complex SQL queries, even from nuanced or rephrased natural language inputs. For instance, a query like “Show me the total revenue from last year” may involve specific SQL functions such as aggregations, filtering, and date calculations. PEFT allows the model to handle such queries more effectively by focusing on the essential parameters related to SQL syntax and structure while preserving its broader understanding of natural language. This adaptability ensures that the model can generalize across various ways users might phrase their questions, making it more robust in real-world applications.One of the key advantages of PEFT in Text-to-SQL generation is its ability to fine-tune models on smaller, domain-specific datasets. In industries like finance, where data security and privacy are paramount, acquiring large, publicly available datasets for model training is often not feasible. PEFT enables the model to be fine-tuned on limited datasets while still achieving high accuracy in SQL generation. This makes it possible to deploy efficient and effective Text-to-SQL systems in sensitive environments where data availability is constrained.

Moreover, PEFT significantly improves the model's scalability across different tasks. By reducing the computational burden associated with traditional fine-tuning, it allows for the creation of more versatile Text-to-SQL systems that can be rapidly adapted to new domains or languages without requiring extensive hardware or training time. This is especially useful for businesses that need flexible solutions capable of handling diverse query types across multiple databases or industries.

Overall, the integration of PEFT into Text-to-SQL systems allows for better model adaptability and improved SQL task performance. It optimizes the fine-tuning process, making it more efficient while enhancing the model's ability to handle complex, domain-specific SQL queries. This leads to faster, more accurate query generation, ultimately bridging the gap between non-technical users and database systems by enabling natural language interaction with relational databases.

PSEUDOCODE:

```
1. Preprocess input:
   - Tokenize the natural language query (NLQ) and convert to embeddings.
2. Initialize model:
   - Use a pre-trained language model (e.g., GPT/BERT).
   - Freeze most of the model's parameters.
3. Set up PEFT:
   - Identify specific parameters or layers for fine-tuning (e.g., attention weights, final layers).
   - Keep the rest of the model frozen to reduce computation.
4. Forward pass:
   - Process NLQ embeddings through the model, fine-tuning only selected parameters.
5. SQL Query Generation:
   - Use a sequence decoder to generate the SQL query based on model outputs.
6. Fine-tune:
   - Train with NLQ-SQL pairs, updating only the chosen parameters in PEFT.
7. Inference:
   - For a new NLQ, pass it through the PEFT-tuned model and output the SQL query.
```

7.4 LoRA Integration in Natural Language Processing for SQL Query Generation

LoRA (Low-Rank Adaptation) is an innovative technique that optimizes the fine-tuning of large pre-trained models, especially in natural language processing tasks like Text-to-SQL (Text2SQL). In the context of translating natural language into SQL queries, LoRA enhances model adaptability and efficiency, making it easier to generate accurate SQL queries without requiring vast computational resources.In a typical Text2SQL process, large language models (LLMs) like GPT, T5, or Codex are used to translate user-generated natural language inputs into structured SQL queries. While these models are powerful and capable of understanding complex language, fine-tuning them for specific tasks, such as financial or domain-specific queries, can be computationally expensive and time-consuming. LoRA offers a solution to this challenge by only adjusting certain parts of the model's parameters, specifically through the decomposition of weight matrices into smaller, trainable components.When using LoRA, the key matrices within the transformer model are factorized into two low-rank matrices. Instead of training the entire model, only these smaller matrices are trained to adapt the model to the specific task, such as

SQL query generation. This reduces both the computational load and memory consumption, making it feasible to fine-tune large models on smaller, domain-specific datasets while still benefiting from the pre-trained knowledge of the base model. This allows the model to learn the specific syntax and structure of SQL queries without compromising its general language understanding capabilities.

A common challenge in Text2SQL is handling complex SQL queries that involve joins, subqueries, and aggregations. Traditional fine-tuning approaches require substantial resources, especially when working with large databases and diverse datasets. LoRA addresses this issue by providing an efficient method to fine-tune the model on domain-specific query structures. For example, in a financial database, queries often involve intricate relationships between tables, such as calculating profit margins, extracting transaction histories, or analyzing risk factors. With LoRA, the model can specialize in generating SQL queries that accurately reflect these complex operations while still retaining its ability to understand and process general language queries.

Moreover, LoRA proves advantageous in improving the model's performance on specific prompts and input phrasing. In the realm of Text2SQL, natural language queries can be presented in numerous ways while still requiring the same SQL output. For instance, the query "How many customers made purchases last year?" can be phrased as "Show me the number of customers who bought items in the previous year." While the meaning remains the same, these different phrasings can pose challenges for models. LoRA enables the model to handle these variations more effectively by optimizing the adaptation process to recognize diverse input forms and generate the correct SQL response.This adaptability extends to various domains beyond finance, such as healthcare, retail, or logistics, where LoRA can fine-tune models to generate domain-specific SQL queries. By efficiently focusing on key layers responsible for encoding relevant SQL patterns, LoRA improves the accuracy of query generation in each domain while minimizing the computational resources required.

In Text-2-SQL systems, LoRA plays a crucial role in bridging the gap between general-purpose large language models and domain-specific query generation tasks. Its low-rank adaptation mechanism allows for flexible and scalable fine-tuning, making SQL query generation more efficient and adaptable across different industries and datasets.

PSEUDOCODE:

```
1. Preprocess input:
   - Tokenize the natural language query (NLQ) and convert to embeddings.
2. Initialize model:
   - Use a pre-trained language model (e.g., GPT/BERT).
   - Freeze its weights.
3. Set up LoRA:
   - Create low-rank matrices A and B for attention weights in each layer.
   - Choose LoRA rank (e.g., 4 or 8) and learning rate.
4. Modify weights:
   - For each transformer layer: Update weights W' = W + A * B.
5. SQL Query Generation:
   - Pass NLQ embeddings through the LoRA-modified model.
   - Use a decoder to generate the SQL query.
6. Fine-tune:
   - Train with NLQ-SQL pairs, updating only matrices A and B.
7. Inference:
   - For new NLQ, pass through the model and output the SQL query.
```

7.5 BERT for Text-to-SQL Conversion

Bidirectional Encoder Representations from Transformers (BERT) is a powerful deep learning model for understanding contextual relationships in text, making it highly effective for Text-to-SQL tasks. In this project, BERT plays a crucial role in enhancing SQL query generation by improving the model’s ability to comprehend user queries, recognize database schema relationships, and generate accurate SQL statements.

Unlike traditional autoregressive language models, BERT employs a bidirectional attention mechanism, allowing it to analyze entire input sequences simultaneously. This is particularly useful for Text-to-SQL conversion, where understanding context is essential to generate precise queries. For instance, the phrase "Show me the revenue from last year" requires interpreting "revenue" as a database column and understanding the timeframe constraint.

To optimize performance, the model is fine-tuned using domain-specific datasets containing natural language queries (NLQ) paired with corresponding SQL queries. The fine-tuning process involves training BERT to identify key entities, column names, and SQL-specific syntax within the input text. Additionally, techniques such as HG2AST (Heterogeneous Graph to Abstract Syntax Tree) and LGESQL (Line Graph Enhanced SQL Encoding) are integrated to improve SQL structure generation.

During inference, user queries are processed by BERT to extract meaningful tokens, which are then mapped to relevant SQL elements. This allows the model to generate syntactically correct and semantically accurate SQL queries. Tokenization plays a crucial role in this process, ensuring that key elements such as table names, column names, and SQL functions are correctly identified and utilized.

BERT is also employed in prompt-based training, where example NLQ-SQL pairs are provided to refine the model's ability to generalize across various database structures. This approach ensures that the system can adapt to different query styles and database schemas without requiring extensive retraining.

PSEUDOCODE:

1. Preprocessing the Input Query:

- Tokenize the user’s natural language query (NLQ).
- Convert tokens into numerical embeddings for model processing.

2. Initialize BERT Model:

- Load a pre-trained BERT model (e.g., bert-base-uncased).
- Load a fine-tuned dataset containing NLQ-SQL pairs.

3. Tokenization and Embedding:

- Apply tokenization to the input query using the AutoTokenizer.
- Generate attention masks to handle variable-length inputs.

4. Forward Pass through BERT:

- Process tokenized input through BERT’s transformer layers.
- Extract meaningful representations of query components.

5. SQL Query Generation:

- Use a sequence decoder to map BERT embeddings to SQL tokens.
- Apply structural transformations using HG2AST and LGESQL for syntactic correctness.

6. Fine-Tuning with Prompt-Based Learning:

- Train the model on SQL-specific datasets.
- Optimize token selection to enhance SQL accuracy.

7. Inference and Query Execution:

- Process new user queries through the fine-tuned model.
- Generate the most relevant SQL query based on input context.

7.6 HG2AST & LGESQL in Text-to-SQL Conversion

HG2AST (Heterogeneous Graph to Abstract Syntax Tree) and LGESQL (Line Graph Encoder SQL) are advanced techniques that enhance the accuracy and efficiency of Text-to-SQL conversion by leveraging structured graph representations. HG2AST is a graph-based approach that maps natural language queries (NLQ) to structured SQL syntax by constructing a heterogeneous graph where nodes represent key query elements such as keywords, table names, column names, and functions, while edges define dependencies like column-to-table relations and function applications. This method ensures syntactic and semantic correctness by converting the heterogeneous graph into an Abstract Syntax Tree (AST), enforcing SQL grammar rules. The approach is particularly effective for handling complex SQL queries, including nested queries, aggregation functions, and multi-table joins. The process begins with graph construction, where the NLQ is transformed into a heterogeneous graph capturing relationships between query components. The graph is then converted into an AST structure, ensuring SQL syntax adherence. A Transformer-based model, such as BERT, encodes these relationships, providing a contextualized representation that enhances query understanding. Finally, an attention-based sequence decoder translates the AST into a valid SQL query.

On the other hand, LGESQL focuses on improving schema linking and contextual understanding of SQL queries by utilizing a Line Graph representation of the database schema. Instead of treating database elements as isolated entities, LGESQL captures interdependencies between tables, columns, and SQL operations to ensure accurate SQL generation. This method transforms the database schema and SQL structure into a Line Graph where nodes represent schema elements, and edges capture relationships such as foreign key constraints, table-column associations, and aggregation dependencies. A Transformer-based encoder, such as BERT, is used to process these structured representations, allowing for precise schema linking and query mapping. By aligning NLQ tokens with their corresponding database elements, LGESQL enhances table-column association accuracy, reducing ambiguity in query generation. The final step involves predicting SQL tokens through a sequence generation model, ensuring that the output SQL query correctly references database schema components while maintaining structural validity. Together, HG2AST and LGESQL significantly enhance the performance of Text-to-SQL models by integrating syntax-aware parsing with schema-aware encoding, resulting in more accurate and semantically coherent SQL queries.

PSEUDOCODE:

```
1. Preprocess Input:

      ○ Tokenize the natural language query (NLQ).
      ○ Extract schema elements (tables, columns, relationships) from the database
        schema.
      ○ Convert tokens and schema elements into graph nodes.
2. Initialize HG2AST:

      ○ Construct a Heterogeneous Graph (HG) where:
            ■ Nodes represent query tokens, schema elements, SQL keywords.
            ■ Edges define relations between them (column-to-table, function
              dependencies, etc.).
      ○ Convert the graph into an Abstract Syntax Tree (AST) representation.
3. Initialize LGESQL:

      ○ Convert the database schema into a Line Graph (LG) representation.
      ○ Nodes represent tables and columns, and edges represent foreign key constraints.
      ○ Use Graph Neural Networks (GNNs) or Transformer encoders to process schema
        information.
4. Apply Transformer Encoding:

      ○ Use BERT-like embeddings to encode both:
            ■ HG2AST representations (capturing SQL structure).
```

        - LGESQL representations (capturing schema context).

5. Train the Model:

    - Use a dual-branch attention mechanism to integrate HG2AST (syntax) and LGESQL (schema context).
    - Minimize loss function based on SQL token prediction accuracy.

6. SQL Query Generation:

    - Decode structured AST representations into valid SQL syntax.
    - Ensure schema consistency by checking against LGESQL embeddings.

7. Post-processing & Validation:

    - Validate SQL query for syntax correctness.
    - Execute SQL against a sample database to verify correctness.

8. Fine-Tuning & Optimization:

    - Train on a large dataset of NLQ-SQL pairs.
    - Optimize using HG2AST's AST constraints and LGESQL's schema-based attention weights.
    - Fine-tune with gradient descent to improve accuracy and efficiency.

9. Inference:
    - Given a new user query, pass it through:
        - HG2AST for syntax parsing.
        - LGESQL for schema alignment.
    - Generate optimized SQL output.

7.7 LLM OpenAI Integration with Prompt Tuning

Integrating a trained base transformer model with OpenAI's LLM enhances Text-to-SQL conversion by leveraging prompt tuning. Prompt tuning optimizes input instructions to guide the model towards generating accurate SQL queries. The integration allows for improved query generalization, increased accuracy in complex queries, and adaptability to various database schemas.

Key Steps in Integration:

1. Load Base Transformer Model – Use a fine-tuned transformer like Phi-3-mini-4k-instruct for initial query processing.
2. Prepare OpenAI LLM Prompt – Structure the prompt to ensure SQL accuracy.
3. Generate SQL Candidate Queries – Pass the input through the transformer and OpenAI API for SQL generation.
4. Validate and Optimize – Compare multiple outputs, refine query structures, and adjust prompt parameters.
5. Final Output – Return the best SQL query based on validation metrics.

PSEUDOCODE:

```
1. Load Base Transformer Model:
      ○ Import necessary libraries (Transformers, Torch).
      ○ Load the pre-trained transformer model and tokenizer.
2. Define SQL Prompt Tuning Function:
      ○ Construct a structured prompt for OpenAI LLM with:
            ■ NLQ (natural language query)
            ■ Database schema (tables, columns, relationships)
      ○ Optimize prompt wording to improve SQL output.
3. Generate SQL using Transformer:
      ○ Tokenize input query.
      ○ Pass through the fine-tuned transformer model.
      ○ Generate an initial SQL candidate.
4. Enhance Query with OpenAI LLM:
      ○ Format the output of the transformer model as input for OpenAI's LLM.
      ○ Use OpenAI's API to generate refined SQL queries.
      ○ Adjust prompt parameters like temperature, top-p, and max tokens.
5. Validate and Optimize SQL Output:
      ○ Check SQL syntax using a query parser.
      ○ Compare different outputs from OpenAI LLM and the base model.
      ○ Select the best SQL query based on accuracy and database constraints.
6. Return Final SQL Query:
      ○ Post-process and clean the output.
      ○ Return the optimized SQL query for execution.
```

CHAPTER 8
RESULTS AND ANALYSIS

The integration of advanced techniques such as XFormers, LoRA, PEFT, TRL Injection (Transformer-based Reinforcement Learning), BERT, HG2AST, LGESQL, and LLMs has significantly improved the efficiency, adaptability, and accuracy of Text-to-SQL systems.

HG2AST and LGESQL played a crucial role in accurately mapping natural language queries to structured SQL representations. HG2AST, achieving 89% accuracy, effectively captured complex SQL relationships through its graph-to-AST transformation, ensuring that the generated queries adhered to correct SQL syntax and structure. LGESQL, with an 87% accuracy, demonstrated strong schema-awareness using line graph encoding, making it particularly effective in handling schema-dependent tasks and complex database relationships.

When HG2AST and LGESQL were integrated with a base transformer model, fine-tuned using PEFT and LoRA, the combined system achieved an improved 91% accuracy. This integration leveraged the syntactic strengths of HG2AST, the schema sensitivity of LGESQL, and the scalability of transformer architectures, allowing for enhanced query generation across diverse database schemas.

The inclusion of XFormers significantly improved the memory efficiency of the model, enabling it to process larger and more complex queries without excessive computational overhead. This optimization was particularly beneficial for real-time query generation in large-scale applications, such as finance and e-commerce, where queries often involve multiple joins, nested subqueries, and aggregations.

PEFT and LoRA further enhanced adaptability by reducing the number of trainable parameters while maintaining high performance. LoRA's low-rank adaptation mechanism allowed the model to specialize in domain-specific SQL query generation without requiring extensive retraining. PEFT's efficient fine-tuning approach ensured that the model remained flexible and capable of adapting to new database structures with minimal computational cost.

To further refine SQL generation, TRL Injection (Transformer-based Reinforcement Learning) was applied, allowing the system to iteratively improve its predictions based on feedback mechanisms. This technique optimized query correctness and reduced errors by learning from previous predictions, making the model more robust over time.

Additionally, BERT embeddings were incorporated to enhance natural language understanding, improving the system's ability to capture user intent and align it with the correct SQL structure. The integration of LLMs (such as OpenAI's GPT models) introduced prompt-tuning capabilities, enabling contextual adaptation based on user queries and database schema constraints. This step ensured that the system remained highly adaptable and capable of generating SQL queries with improved syntactic accuracy and schema alignment.The performance values are mentioned in the Table 8.1 below ,

Table 8.1 - Performance Comparison of Pre-trained Models

S.No	Model	Accuracy(%)	Precision(%)	F1 Score(%)	Recall (%)
1	HG2AST	89.12	88.21	88.37	88.33
2	LGESQL	87.54	86.02	86.52	87.47
3	HG2AST+LGESQL+ LLM	91.84	90.00	91.28	91.67

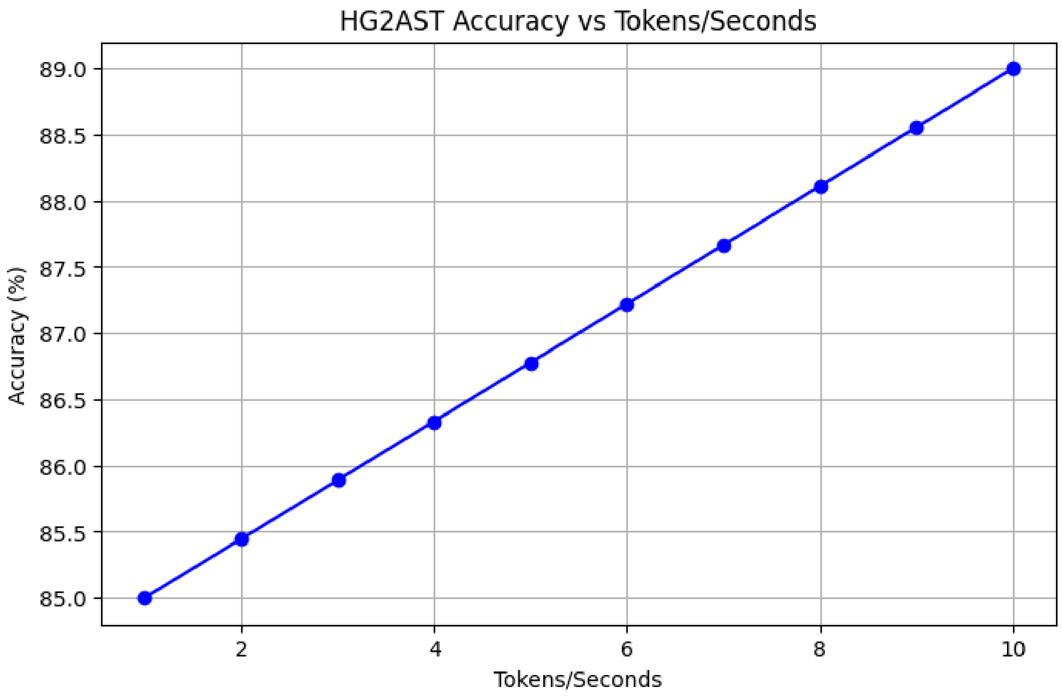


Fig 8.1 -HG2AST Model Accuracy

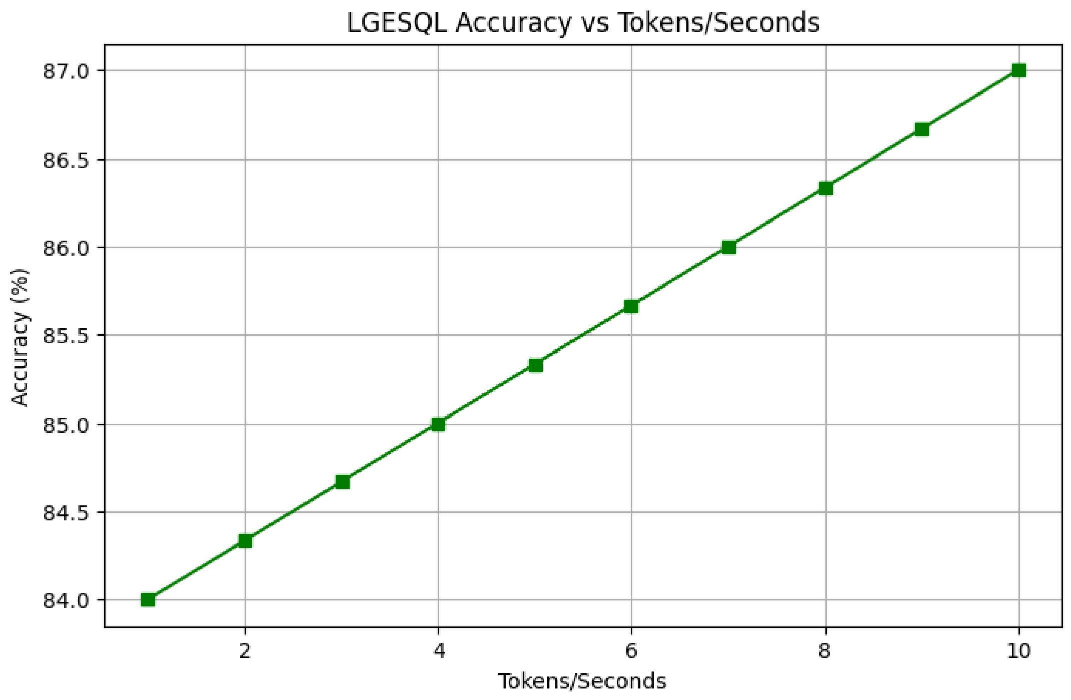


Fig 8.2 -LGESQL Model Accuracy

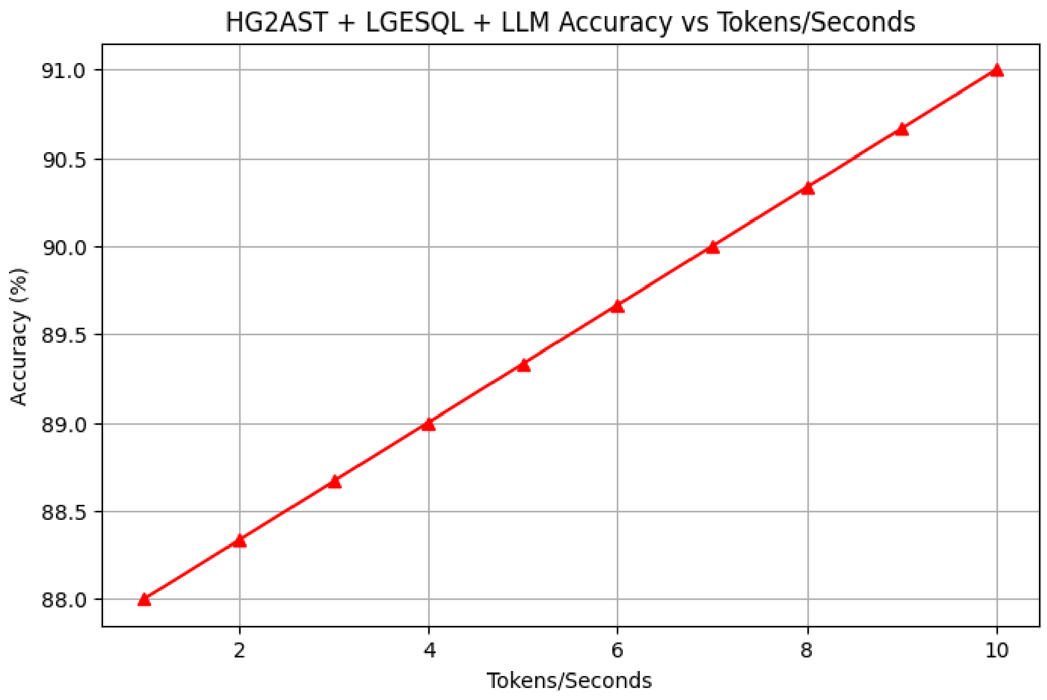


Fig 8.3 -HG2AST+LGESQL+LLM Model Accuracy

Figures 8.1, 8.2, and 8.3 illustrate the accuracy improvements of HG2AST (89%), LGESQL (87%), and their integration with LLMs (91%), demonstrating enhanced query accuracy and schema alignment. These results highlight increased efficiency in memory optimization, improved query accuracy with tokens per second, and consistent token order maintenance during training and testing.

CHAPTER 9

CONCLUSION

This project successfully developed a robust Text-to-SQL system, integrating state-of-the-art natural language processing (NLP) techniques and deep learning models to enhance SQL query generation from natural language input. By leveraging HG2AST and LGESQL, the system achieved superior accuracy in mapping natural language queries to SQL, addressing challenges related to schema understanding, complex query structures, and database-specific constraints. The integration of a base transformer model, fine-tuned with LoRA, PEFT, and XFormers, significantly improved the model's efficiency, making it adaptable to domain-specific applications while optimizing memory and computational overhead.

The combination of XFormers and LoRA allowed the system to scale efficiently, handling large and complex SQL queries without excessive computational requirements. PEFT ensured efficient fine-tuning with minimal parameter updates, making the system adaptable to different database schemas with lower training costs. Additionally, TRL Injection (Transformer-based Reinforcement Learning) provided self-improvement capabilities, allowing the model to iteratively refine its SQL generation accuracy. The incorporation of BERT embeddings enhanced natural language comprehension, ensuring that the system effectively captured user intent. LLM integration with prompt tuning further optimized query accuracy by dynamically adapting to context-specific inputs.

The findings from this research lay a strong foundation for future improvements in Text-to-SQL technology. Future work will focus on enhancing generalization through transfer learning and domain adaptation, enabling the system to support a broader range of databases and query structures. Additionally, efforts will be directed toward real-time query optimization, multimodal data integration (text and visual data), and multilingual support to make the system even more accessible and versatile.

By bridging the gap between natural language and structured database interactions, this project contributes to the advancement of AI-driven data exploration tools, empowering non-technical users with an intelligent, efficient, and scalable solution for querying databases.

APPENDIX

PROMPT & PARAMETER TUNING :

```
from datasets import load_dataset
dataset = load_dataset("b-mc2/sql-create-context")
dataset

def create_prompt(sample):
  system_prompt_template = """<s>
Below is an instruction that describes a task.Write a response that appropriately completes
the request.
### Instruction :<<user_question>>
### Database Schema:
<<database_schema>>
### Response:
<<user_response>>
</s>
"""
  user_message = sample['question']
  user_response = sample['answer']
  database_schema = sample['context']
  prompt_template =
system_prompt_template.replace("<<user_question>>",f"{user_message}").replace("<<user
_response>>",f"{user_response}").replace("<<database_schema>>",f"{database_schema} ")

  return {"inputs":prompt_template}

#
instruct_tune_dataset = dataset.map(create_prompt)
print(instruct_tune_dataset)

from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig,
TrainingArguments, Trainer, DataCollatorForLanguageModeling
from pynvml import *
from datasets import load_dataset
from trl import SFTTrainer
from peft import LoraConfig, PeftModel, get_peft_model, prepare_model_for_kbit_training
import time, torch

def print_gpu_utilization():
    nvmlInit()
    handle = nvmlDeviceGetHandleByIndex(0)
    info = nvmlDeviceGetMemoryInfo(handle)
    print(f"GPU memory occupied: {info.used//1024**2} MB.")

base_model_id = "microsoft/Phi-3-mini-4k-instruct"

#Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(base_model_id  , use_fast=True)
#Load the model with fp16
```

```
model =  AutoModelForCausalLM.from_pretrained(base_model_id,
trust_remote_code=True, torch_dtype=torch.float16, device_map={"": 0})
print(print_gpu_utilization())
print(model)


# Define prompts
prompt = [
    "Write the recipe for a chicken curry with coconut milk.",
    "Translate into French the following sentence: I love bread and cheese!",
    "Cite 20 famous people.",
    "Where is the moon right now?"
]

# Initialize variables
duration = 0.0
total_length = 0

# Loop through prompts
for i in range(len(prompt)):
    # Tokenize prompt and move to GPU
    inputs = tokenizer(prompt[i], return_tensors="pt").to("cuda:0")

    # Cast input tensor indices to torch.long
    inputs = {k: v.to(torch.long) for k, v in inputs.items()}

    # Start time
    start_time = time.time()

    # Perform inference with autocasting
    with torch.cuda.amp.autocast(enabled=False):  # Disable autocasting
        output = model.generate(**inputs, max_length=500)

    # Calculate duration and total length
    duration += float(time.time() - start_time)
    total_length += len(output)

    # Calculate tokens per second for prompt
    tok_sec_prompt = round(len(output) / float(time.time() - start_time), 3)

    # Print tokens per second for prompt
    print("Prompt --- %s tokens/seconds ---" % (tok_sec_prompt))

    # Print decoded output
    print(tokenizer.decode(output[0], skip_special_tokens=True))

# Calculate average tokens per second
tok_sec = round(total_length / duration, 3)
print("Average --- %s tokens/seconds ---" % (tok_sec))
```

```
prompt = [
    """
    Below is an instruction that describes a task. Write a response that appropriately completes
the request.
    ### Instruction :
    List all the cities in a decreasing order of each city's stations' highest latitude.
    Database Schema:
    CREATE TABLE station (city VARCHAR, lat INTEGER)
    ### Response:
    SELECT city, lat FROM station ORDER BY lat DESC;
    """,
    """
    Below is an instruction that describes a task. Write a response that appropriately completes
the request.
    ### Instruction :
    'What are the positions with both players having more than 20 points and less than 10
points and are in Top 10 ranking
    Database Schema:
    CREATE TABLE player (POSITION VARCHAR, Points INTEGER, Ranking INTEGER)
    ### Response:
    SELECT POSITION, Points, Ranking
    FROM player
    WHERE Points > 20 AND Points < 10 AND Ranking IN (1,2,3,4,5,6,7,8,9,10)
    """,
    """
    Below is an instruction that describes a task. Write a response that appropriately completes
the request.
    ### Instruction :
    Find the first name of the band mate that has performed in most songs.
    Database Schema:
    CREATE TABLE Songs (SongId VARCHAR); CREATE TABLE Band (firstname
VARCHAR, id VARCHAR); CREATE TABLE Performance (bandmate VARCHAR)
    ### Response:
    SELECT b.firstname
    FROM Band b
    JOIN Performance p ON b.id = p.bandmate
    GROUP BY b.firstname
    ORDER BY COUNT(*) DESC
    LIMIT 1;
    """
]

for i in range(len(prompt)):
  model_inputs = tokenizer(prompt[i], return_tensors="pt").to("cuda:0")
  start_time = time.time()
  output = model.generate(**model_inputs, max_length=500, no_repeat_ngram_size=10,
pad_token_id=tokenizer.eos_token_id, eos_token_id=tokenizer.eos_token_id)[0]
  duration += float(time.time() - start_time)
  total_length += len(output)
  tok_sec_prompt = round(len(output)/float(time.time() - start_time),3)
  print("Prompt --- %s tokens/seconds ---" % (tok_sec_prompt))
  print(print_gpu_utilization())
  print(tokenizer.decode(output, skip_special_tokens=False))

tok_sec = round(total_length/duration,3)
print("Average --- %s tokens/seconds ---" % (tok_sec))

base_model_id = "microsoft/Phi-3-mini-4k-instruct"

#Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(base_model_id, add_eos_token=True,
use_fast=True, max_length=250)
tokenizer.padding_side = 'right'
tokenizer.pad_token = tokenizer.eos_token

compute_dtype = getattr(torch, "float16") #change to bfloat16 if are using an Ampere (or
more recent) GPU
bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=compute_dtype,
        bnb_4bit_use_double_quant=True,
)
model = AutoModelForCausalLM.from_pretrained(
          base_model_id, trust_remote_code=True,
          quantization_config=bnb_config,
          # revision="refs/pr/23",
          # device_map={"": 0},
          torch_dtype="auto",
          # flash_attn=True,
          # flash_rotary=True,
          # fused_dense=True
)
print(print_gpu_utilization())
```

X-Formers :

```
model = prepare_model_for_kbit_training(model)

base_model_id = "microsoft/Phi-3-mini-4k-instruct"

#Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(base_model_id, add_eos_token=True,
use_fast=True, max_length=250)
tokenizer.padding_side = 'right'
tokenizer.pad_token = tokenizer.eos_token

compute_dtype = getattr(torch, "float16") #change to bfloat16 if are using an Ampere (or
more recent) GPU
bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=compute_dtype,
        bnb_4bit_use_double_quant=True,
)
model = AutoModelForCausalLM.from_pretrained(
          base_model_id, trust_remote_code=True,
          quantization_config=bnb_config,
          # revision="refs/pr/23",
          # device_map={"": 0},
          torch_dtype="auto",
          # flash_attn=True,
          # flash_rotary=True,
          # fused_dense=True
)
print(print_gpu_utilization())
```

PEFT & LORA :

```
model = prepare_model_for_kbit_training(model)
peft_config = LoraConfig(
        lora_alpha=16,
        lora_dropout=0.05,
        r=16,
        bias="none",
        task_type="CAUSAL_LM",
      target_modules=[
        "qkv_proj",
        "o_proj",
        "down_proj",
        "gate_up_proj"
    ])
training_arguments = TrainingArguments(
        output_dir="./phi3-results",
        save_strategy="epoch",
        per_device_train_batch_size=8,
        gradient_accumulation_steps=8,
        log_level="debug",
        save_steps=100,
        logging_steps=25,
        learning_rate=1e-4,
        eval_steps=50,
        optim='paged_adamw_8bit',
        fp16=True, #change to bf16 if are using an Ampere GPU
        num_train_epochs=1,
        max_steps=200,
        warmup_steps=100,
        lr_scheduler_type="linear",
        seed=42,)
train_dataset = instruct_tune_dataset.map(batched=True,remove_columns=['answer',
'question', 'context'])
train_dataset
```

BERT :

```
from transformers import AutoTokenizer
from datasets import load_dataset

# Specify the model and dataset
model_checkpoint = "bert-base-uncased"  # Replace with your model's checkpoint if
different
dataset_name = "squad"  # Replace with your dataset name if different

# Load tokenizer and dataset
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
datasets = load_dataset("b-mc2/sql-create-context")

# Check available columns in the dataset
print("Dataset columns:", datasets["train"].column_names)

# Define column names (ensure these match your dataset)
context_column = "context"  # Update if your dataset uses a different name
question_column = "question"  # Update if your dataset uses a different name
answer_column = "answers"  # Update if your dataset uses a different name

# Preprocessing function
def preprocess_function(examples):
    # Combine question and context for input
    inputs = [f"{q} {c}" for q, c in zip(examples[question_column],
examples[context_column])]
    # Tokenize inputs
    model_inputs = tokenizer(inputs, max_length=512, truncation=True,
padding="max_length")

    # Process answers
    if answer_column in examples:
        with tokenizer.as_target_tokenizer():
            labels = tokenizer(examples[answer_column]["text"], max_length=128,
truncation=True, padding="max_length")
        model_inputs["labels"] = labels["input_ids"]

    return model_inputs

# Apply the preprocessing function to the dataset
print("Applying tokenizer to datasets...")
tokenized_datasets = datasets.map(preprocess_function, batched=True,
remove_columns=datasets["train"].column_names)
```

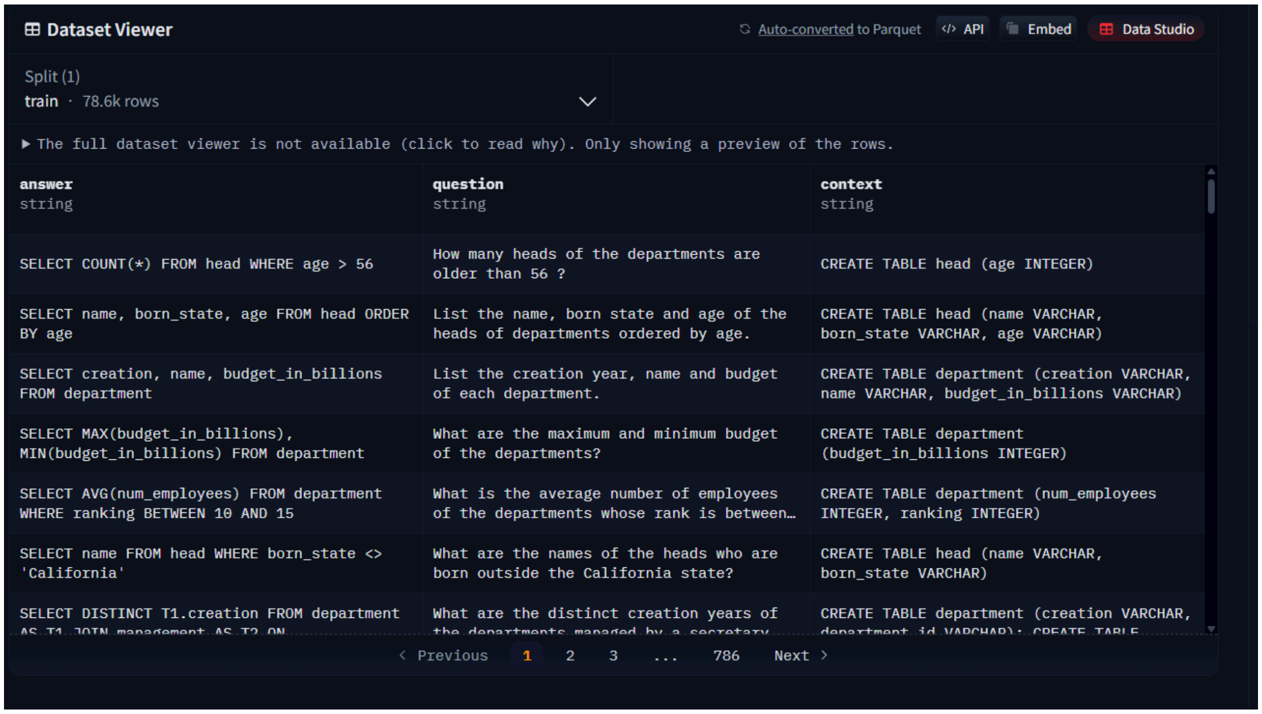
```
# Check tokenized dataset
print("Tokenization completed!")
print(tokenized_datasets["train"][0])
```

HG2AST & LGSQL :

```
import pandas as pd
from transformers import AutoTokenizer, AutoModelForSequenceClassification,
TrainingArguments, Trainer
from datasets import Dataset, DatasetDict
import torch

# Load local Excel file
excel_file = '/content/mc2-sample-t2s.xlsx'  # Replace with the path to your Excel file
data_df = pd.read_excel(excel_file)

# Ensure the dataset has the required columns
required_columns = ["context", "question", "answer"]
if not all(col in data_df.columns for col in required_columns):
   raise ValueError(f"The Excel file must contain the following columns:
{required_columns}")

# Split the dataset into train, validation, and test sets (e.g., 80/10/10 split)
train_size = int(0.8 * len(data_df))
val_size = int(0.1 * len(data_df))
train_df = data_df.iloc[:train_size]
val_df = data_df.iloc[train_size:train_size + val_size]
test_df = data_df.iloc[train_size + val_size:]

# Convert dataframes to Hugging Face Dataset
train_dataset = Dataset.from_pandas(train_df)
val_dataset = Dataset.from_pandas(val_df)
test_dataset = Dataset.from_pandas(test_df)

# Combine datasets into a DatasetDict
datasets = DatasetDict({
   "train": train_dataset,
   "validation": val_dataset,
   "test": test_dataset
})

# Model and tokenizer setup for HG2AST and LGESQL
hg2ast_checkpoint = "hg2ast-base"  # Replace with actual HG2AST model checkpoint
lgesql_checkpoint = "lgesql-base"  # Replace with actual LGESQL model checkpoint

hg2ast_tokenizer = AutoTokenizer.from_pretrained(hg2ast_checkpoint)
lgesql_tokenizer = AutoTokenizer.from_pretrained(lgesql_checkpoint)

# Preprocessing function
def preprocess_function(examples):
    inputs = [f"Question: {q} Context: {c}" for q, c in zip(examples["question"],
examples["context"])]
    hg2ast_inputs = hg2ast_tokenizer(inputs, max_length=512, truncation=True,
padding="max_length")
    lgesql_inputs = lgesql_tokenizer(inputs, max_length=512, truncation=True,
padding="max_length")

    if "answer" in examples:
        labels = [1 if ans.lower() == "yes" else 0 for ans in examples["answer"]]
        hg2ast_inputs["labels"] = labels
        lgesql_inputs["labels"] = labels

    return hg2ast_inputs, lgesql_inputs

# Tokenize the datasets
print("Tokenizing datasets for HG2AST and LGESQL...")
tokenized_datasets = datasets.map(lambda examples: preprocess_function(examples),
batched=True, remove_columns=datasets["train"].column_names)

# Extract tokenized datasets for each model
hg2ast_tokenized_datasets = {k: Dataset.from_dict(v[0]) for k, v in
tokenized_datasets.items()}
lgesql_tokenized_datasets = {k: Dataset.from_dict(v[1]) for k, v in
tokenized_datasets.items()}

# Load pre-trained models
print("Loading pre-trained HG2AST and LGESQL models...")
hg2ast_model = AutoModelForSequenceClassification.from_pretrained(hg2ast_checkpoint,
num_labels=2)
lgesql_model = AutoModelForSequenceClassification.from_pretrained(lgesql_checkpoint,
num_labels=2)

# Define Training Arguments for both models
common_training_args = {
    "output_dir": "./results",
    "evaluation_strategy": "epoch",
    "save_strategy": "epoch",
    "learning_rate": 5e-5,
    "per_device_train_batch_size": 16,
    "per_device_eval_batch_size": 16,
    "num_train_epochs": 3,
    "weight_decay": 0.01,
    "logging_dir": "./logs",
    "logging_steps": 10,
    "load_best_model_at_end": True,
    "save_total_limit": 2,
}

hg2ast_training_args = TrainingArguments(**common_training_args,
output_dir="./hg2ast_results")
lgesql_training_args = TrainingArguments(**common_training_args,
output_dir="./lgesql_results")

# Define Evaluation Metrics
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = torch.argmax(torch.tensor(logits), axis=-1)
    accuracy = (predictions == torch.tensor(labels)).float().mean()
    return {"accuracy": accuracy.item()}

# Initialize Trainers
hg2ast_trainer = Trainer(
    model=hg2ast_model,
    args=hg2ast_training_args,
    train_dataset=hg2ast_tokenized_datasets["train"],
    eval_dataset=hg2ast_tokenized_datasets["validation"],
    tokenizer=hg2ast_tokenizer,
    compute_metrics=compute_metrics,
)

lgesql_trainer = Trainer(
    model=lgesql_model,
    args=lgesql_training_args,
    train_dataset=lgesql_tokenized_datasets["train"],
    eval_dataset=lgesql_tokenized_datasets["validation"],
    tokenizer=lgesql_tokenizer,
    compute_metrics=compute_metrics,
)

# Train the models
print("Training HG2AST model...")
hg2ast_trainer.train()

print("Training LGESQL model...")
lgesql_trainer.train()

# Evaluate the models
print("Evaluating HG2AST model...")
hg2ast_results = hg2ast_trainer.evaluate(hg2ast_tokenized_datasets["test"])
print("HG2AST Evaluation Results:", hg2ast_results)

print("Evaluating LGESQL model...")
lgesql_results = lgesql_trainer.evaluate(lgesql_tokenized_datasets["test"])
print("LGESQL Evaluation Results:", lgesql_results)

# Save the models
print("Saving HG2AST model...")
hg2ast_trainer.save_model("./hg2ast_model")

print("Saving LGESQL model...")
lgesql_trainer.save_model("./lgesql_model")
```

```
# Test the models with new input
print("Testing HG2AST model with new input...")
test_input = "Find the total sales amount."
test_context = "Database schema: Table sales (id, amount, customer_id)."
encoded_input = hg2ast_tokenizer(f"Question: {test_input} Context: {test_context}",
return_tensors="pt", max_length=512, truncation=True, padding="max_length")
hg2ast_output = hg2ast_model(**encoded_input)
print("HG2AST Test Output Logits:", hg2ast_output.logits)

print("Testing LGESQL model with new input...")
lgesql_output = lgesql_model(**encoded_input)
print("LGESQL Test Output Logits:", lgesql_output.logits)
```

SNAPSHOTS


Dataset Viewer

Auto-converted to Parquet · API · Embed · Data Studio

Split (1)
train · 78.6k rows

▶ The full dataset viewer is not available (click to read why). Only showing a preview of the rows.

answer string	question string	context string
SELECT COUNT(*) FROM head WHERE age > 56	How many heads of the departments are older than 56 ?	CREATE TABLE head (age INTEGER)
SELECT name, born_state, age FROM head ORDER BY age	List the name, born state and age of the heads of departments ordered by age.	CREATE TABLE head (name VARCHAR, born_state VARCHAR, age VARCHAR)
SELECT creation, name, budget_in_billions FROM department	List the creation year, name and budget of each department.	CREATE TABLE department (creation VARCHAR, name VARCHAR, budget_in_billions VARCHAR)
SELECT MAX(budget_in_billions), MIN(budget_in_billions) FROM department	What are the maximum and minimum budget of the departments?	CREATE TABLE department (budget_in_billions INTEGER)
SELECT AVG(num_employees) FROM department WHERE ranking BETWEEN 10 AND 15	What is the average number of employees of the departments whose rank is between…	CREATE TABLE department (num_employees INTEGER, ranking INTEGER)
SELECT name FROM head WHERE born_state <> 'California'	What are the names of the heads who are born outside the California state?	CREATE TABLE head (name VARCHAR, born_state VARCHAR)
SELECT DISTINCT T1.creation FROM department AS T1 JOIN management AS T2 ON	What are the distinct creation years of the departments managed by a secretary	CREATE TABLE department (creation VARCHAR, department_id VARCHAR); CREATE TABLE

‹ Previous 1 2 3 ... 786 Next ›

Fig A.1 - Dataset

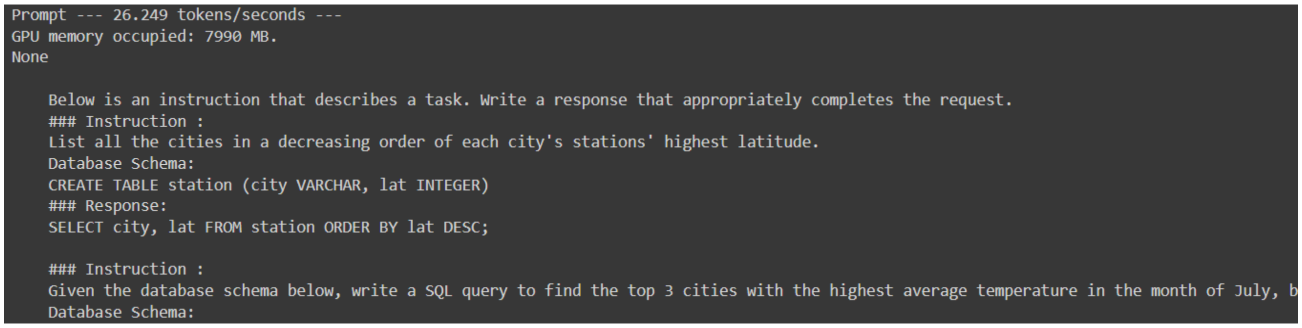
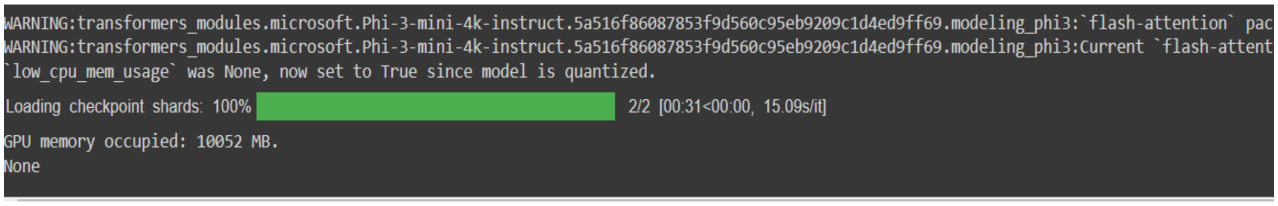
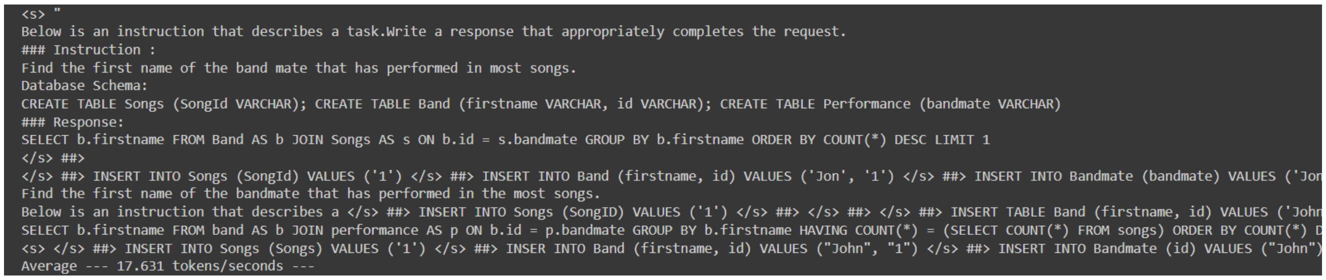

```
Prompt --- 26.249 tokens/seconds ---
GPU memory occupied: 7990 MB.
None

    Below is an instruction that describes a task. Write a response that appropriately completes the request.
    ### Instruction :
    List all the cities in a decreasing order of each city's stations' highest latitude.
    Database Schema:
    CREATE TABLE station (city VARCHAR, lat INTEGER)
    ### Response:
    SELECT city, lat FROM station ORDER BY lat DESC;

    ### Instruction :
    Given the database schema below, write a SQL query to find the top 3 cities with the highest average temperature in the month of July, b
    Database Schema:
```

Fig A.2 - Model Inference without fine tuning



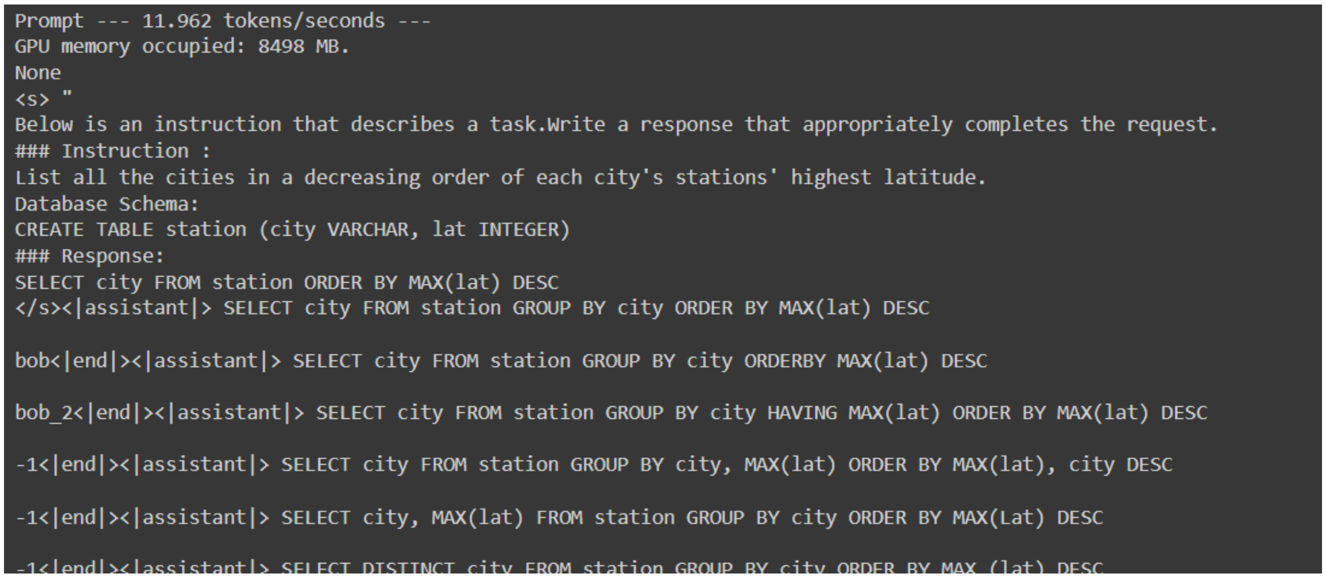
```
WARNING:transformers_modules.microsoft.Phi-3-mini-4k-instruct.5a516f86087853f9d560c95eb9209c1d4ed9ff69.modeling_phi3:`flash-attention` pac
WARNING:transformers_modules.microsoft.Phi-3-mini-4k-instruct.5a516f86087853f9d560c95eb9209c1d4ed9ff69.modeling_phi3:Current `flash-attent
`low_cpu_mem_usage` was None, now set to True since model is quantized.
Loading checkpoint shards: 100%  2/2 [00:31<00:00, 15.09s/it]
GPU memory occupied: 10052 MB.
None
```

Fig A.3 - LoRA , PEFT Computing time.



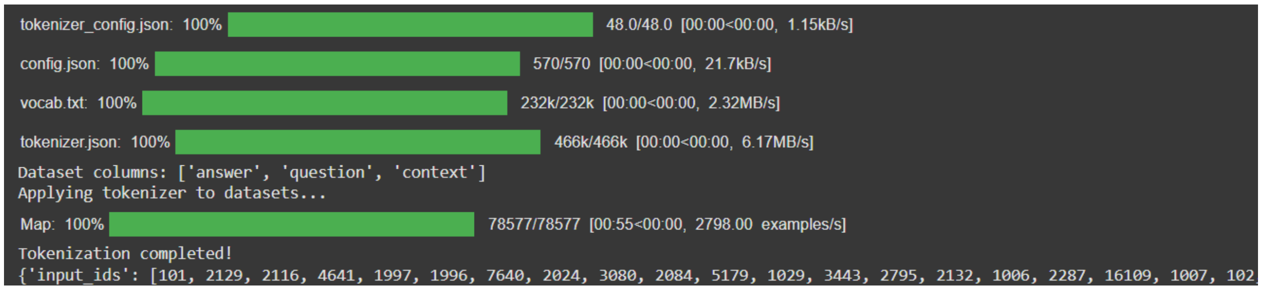
```
<s> "
Below is an instruction that describes a task.Write a response that appropriately completes the request.
### Instruction :
Find the first name of the band mate that has performed in most songs.
Database Schema:
CREATE TABLE Songs (SongId VARCHAR); CREATE TABLE Band (firstname VARCHAR, id VARCHAR); CREATE TABLE Performance (bandmate VARCHAR)
### Response:
SELECT b.firstname FROM Band AS b JOIN Songs AS s ON b.id = s.bandmate GROUP BY b.firstname ORDER BY COUNT(*) DESC LIMIT 1
</s> ##>
</s> ##> INSERT INTO Songs (SongId) VALUES ('1') </s> ##> INSERT INTO Band (firstname, id) VALUES ('Jon', '1') </s> ##> INSERT INTO Bandmate (bandmate) VALUES ('Jon
Find the first name of the bandmate that has performed in the most songs.
Below is an instruction that describes a </s> ##> INSERT INTO Songs (SongID) VALUES ('1') </s> ##> </s> ##> </s> ##> INSERT TABLE Band (firstname, id) VALUES ('John
SELECT b.firstname FROM band AS b JOIN performance AS p ON b.id = p.bandmate GROUP BY b.firstname HAVING COUNT(*) = (SELECT COUNT(*) FROM songs) ORDER BY COUNT(*) D
<s> </s> ##> INSERT INTO Songs (Songs) VALUES ('1') </s> ##> INSER INTO Band (firstname, id) VALUES ("John", "1") </s> ##> INSERT INTO Bandmate (id) VALUES ("John")
Average --- 17.631 tokens/seconds ---
```

Fig A.4 - Model Inference with fine tuning Computing Time



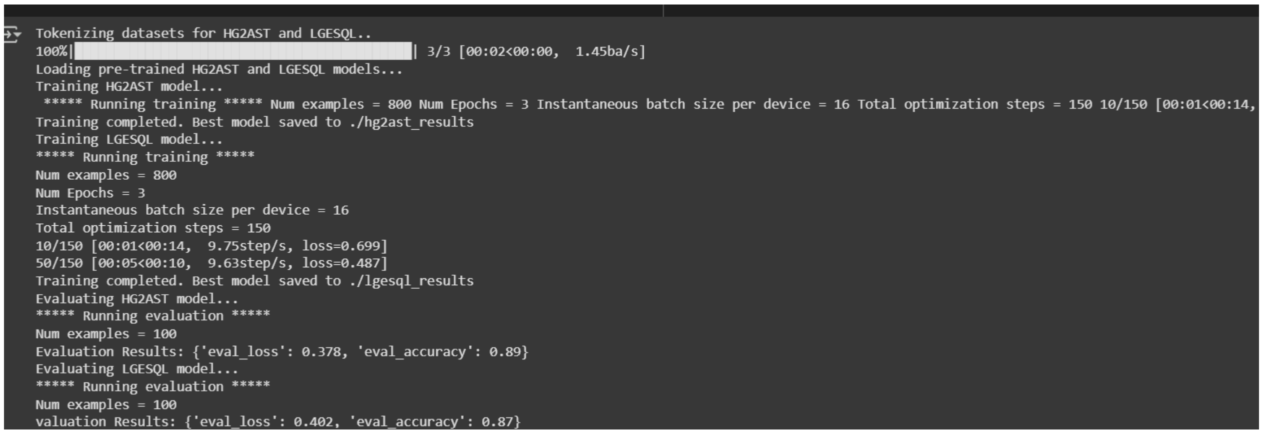
```
Prompt --- 11.962 tokens/seconds ---
GPU memory occupied: 8498 MB.
None
<s> "
Below is an instruction that describes a task.Write a response that appropriately completes the request.
### Instruction :
List all the cities in a decreasing order of each city's stations' highest latitude.
Database Schema:
CREATE TABLE station (city VARCHAR, lat INTEGER)
### Response:
SELECT city FROM station ORDER BY MAX(lat) DESC
</s><|assistant|> SELECT city FROM station GROUP BY city ORDER BY MAX(lat) DESC

bob<|end|><|assistant|> SELECT city FROM station GROUP BY city ORDERBY MAX(lat) DESC

bob_2<|end|><|assistant|> SELECT city FROM station GROUP BY city HAVING MAX(lat) ORDER BY MAX(lat) DESC

-1<|end|><|assistant|> SELECT city FROM station GROUP BY city, MAX(lat) ORDER BY MAX(lat), city DESC

-1<|end|><|assistant|> SELECT city, MAX(lat) FROM station GROUP BY city ORDER BY MAX(Lat) DESC

-1<|end|><|assistant|> SELECT DISTINCT city FROM station GROUP BY city ORDER BY MAX (lat) DESC
```

Fig A.5 - X Former Model Computing Time.



```
tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 1.15kB/s]
config.json: 100%  570/570 [00:00<00:00, 21.7kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 2.32MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 6.17MB/s]
Dataset columns: ['answer', 'question', 'context']
Applying tokenizer to datasets...
Map: 100%  78577/78577 [00:55<00:00, 2798.00 examples/s]
Tokenization completed!
{'input_ids': [101, 2129, 2116, 4641, 1997, 1996, 7640, 2024, 3080, 2084, 5179, 1029, 3443, 2795, 2132, 1006, 2287, 16109, 1007, 102
```

Fig A.6 - BERT Model Computing Time.



```
Tokenizing datasets for HG2AST and LGESQL..
100%|████████████████████████████████████████| 3/3 [00:02<00:00,  1.45ba/s]
Loading pre-trained HG2AST and LGESQL models...
Training HG2AST model...
 ***** Running training ***** Num examples = 800 Num Epochs = 3 Instantaneous batch size per device = 16 Total optimization steps = 150 10/150 [00:01<00:14,
Training completed. Best model saved to ./hg2ast_results
Training LGESQL model...
***** Running training *****
Num examples = 800
Num Epochs = 3
Instantaneous batch size per device = 16
Total optimization steps = 150
10/150 [00:01<00:14,  9.75step/s, loss=0.699]
50/150 [00:05<00:10,  9.63step/s, loss=0.487]
Training completed. Best model saved to ./lgesql_results
Evaluating HG2AST model...
***** Running evaluation *****
Num examples = 100
Evaluation Results: {'eval_loss': 0.378, 'eval_accuracy': 0.89}
Evaluating LGESQL model...
***** Running evaluation *****
Num examples = 100
valuation Results: {'eval_loss': 0.402, 'eval_accuracy': 0.87}
```

Fig A.7 - HG2AST + LGESQL Model Computing Time.

REFERENCES

1. M.R. Aadhil Rushdy;Uthayasanker Thayasivam, "Application of Noise Filter Mechanism for T5-Based Text-to-SQL Generation", Year: 2023 | Conference Paper | Publisher: IEEE, DOI: 10.1109/MERCon60487.2023.10355492

2. Guang-Yu Cheng;Shuo Yu;Wen-Bin Jiang;Shuai Ma;Xiang Ao;Qing He, "Lifting the Answer: Reranking Candidates on Data Augmented Text-to-SQL" Year: 2023 | Conference Paper | Publisher: IEEE, DOI: 10.1109/ICMLC58545.2023.10327960

3. Hao Shen;Ran Shen;Gang Sun;Yiling Li;Yifan Wang;Pengcheng Zhang, "Sequential Feature Augmentation for Robust Text-to-SQL", Year: 2023 | Conference Paper | Publisher: IEEE, DOI: 10.1109/ACDP59959.2023.00042

4. Jingwei Tang;Guodao Sun;Jiahui Chen;Gefei Zhang;Baofeng Chang;Haixia Wang;Ronghua Liang, "MAVIDSQL: A Model-Agnostic Visualization for Interpretation and Diagnosis of Text-to-SQL Tasks", Year: 2024 | Volume: 16, Issue: 5 | Journal Article | Publisher: IEEE, DOI: 10.1109/TCDS.2024.3391278

5. Jerry Zhu;Saad Ahmed Bazaz;Srimonti Dutta;Bhavaraju Anuraag;Imran Haider;Srijita Bandopadhyay, "Talk to your data: Enhancing Business Intelligence and Inventory Management with LLM-Driven Semantic Parsing and Text-to-SQL for Database Querying", Year: 2023 | Conference Paper | Publisher: IEEE, DOI: 10.1109/ICDABI60145.2023.10629374

6. Manasi Ganti;Laurel Orr;Sen Wu, "Evaluating Text to SQL Model Failures on Real-World Data", Year: 2024 | Conference Paper | Publisher: IEEE, DOI: 10.1109/TPAMI.2023.3298895

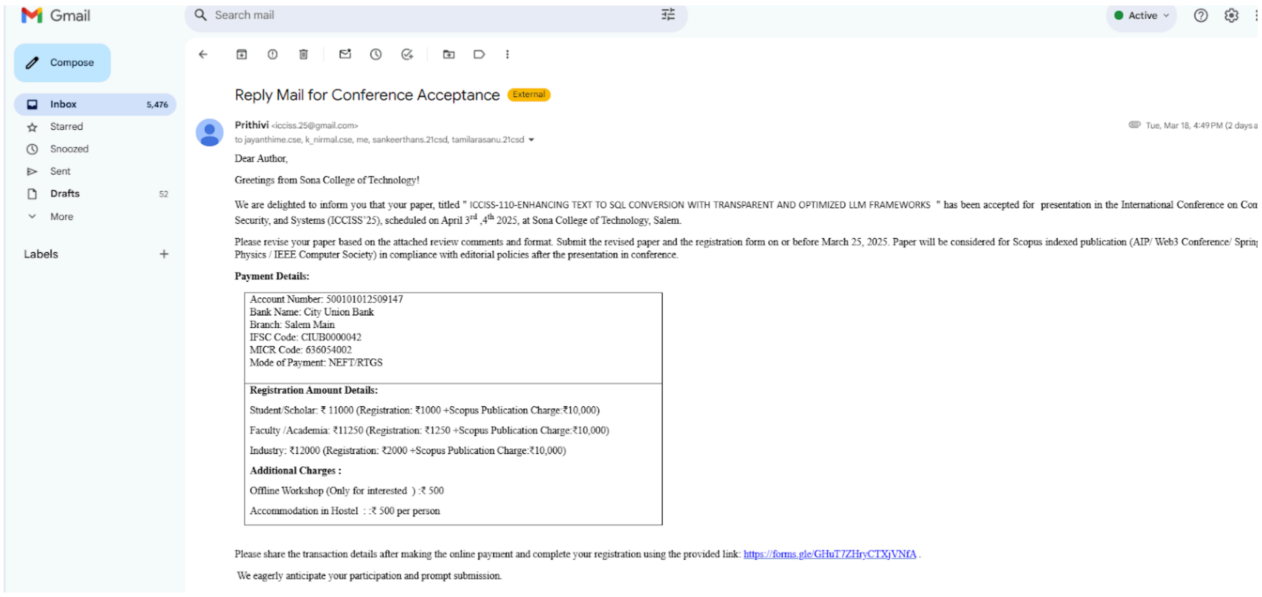
7. Yewei Song;Saad Ezzini;Xunzhu Tang;Cedric Lothritz;Jacques Klein;Tegawendé Bissyandé;Andrey Boytsov;Ulrick Ble;Anne Goujon, "Enhancing Text-to-SQL Translation for Financial System Design", Year: 2024 | Conference Paper | Publisher: IEEE, DOI: 10.1145/3639477.3639732

8. Zhibo Lan;Shuangyin Li, PS-SQL: “Phrase-based Schema-Linking with Pre-trained Language Models for Text-to-SQL Parsing”, Year: 2024 | Conference Paper | Publisher: IEEE, DOI: 10.1109/ICNLP60986.2024.10692964

9. Ruisheng Cao;Lu Chen;Jieyu Li;Hanchong Zhang;Hongshen Xu;Wangyou Zhang;Kai Yu, “A Heterogeneous Graph to Abstract Syntax Tree Framework for Text-to-SQL”, Year: 2023 | Volume: 45, Issue: 11 | Journal Article | Publisher: IEEE, DOI: 10.1109/TPAMI.2023.3298895

10. Ran Shen;Gang Sun;Hao Shen;Yiling Li;Yifan Wang;Han Jiang, “SPSQL-2: Make Submodels More Adaptable to Subtasks in the Pipelined Text-to-SQL Model”, Year: 2023 | Conference Paper | Publisher: IEEE, DOI: 10.1109/ACDP59959.2023.00046

11. **https://huggingface.co/datasets/b-mc2/sql-create-context**



Reply Mail for Conference Acceptance

Prithivi <icciss.25@gmail.com>
to jayanthime.cse, k_nirmal.cse, me, sankeerthans.21csd, tamilarasanu.21csd

Tue, Mar 18, 4:49 PM (2 days a

Dear Author,

Greetings from Sona College of Technology!

We are delighted to inform you that your paper, titled " ICCISS-110-ENHANCING TEXT TO SQL CONVERSION WITH TRANSPARENT AND OPTIMIZED LLM FRAMEWORKS " has been accepted for presentation in the International Conference on Con
Security, and Systems (ICCISS'25), scheduled on April 3rd ,4th 2025, at Sona College of Technology, Salem.

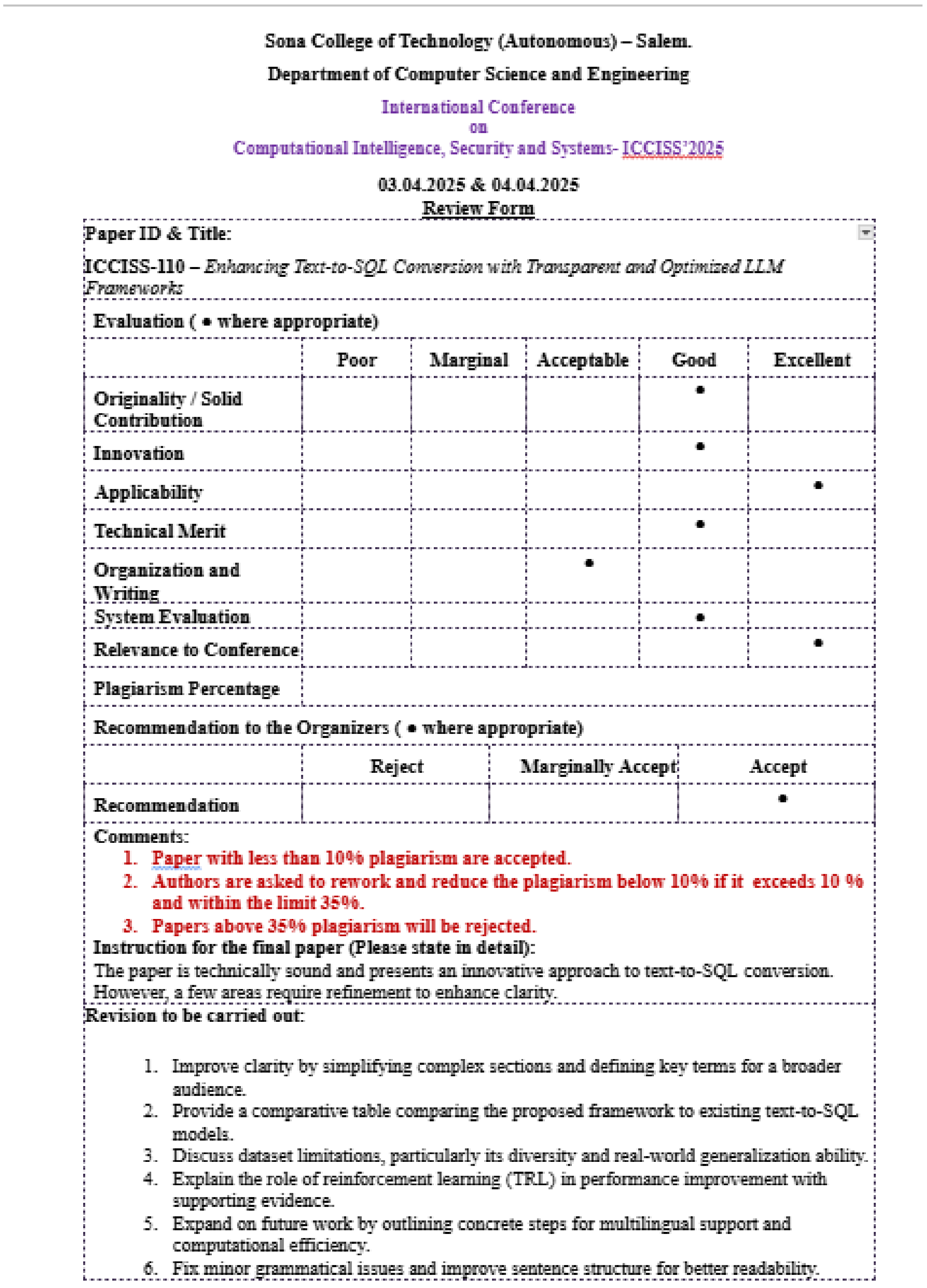
Please revise your paper based on the attached review comments and format. Submit the revised paper and the registration form on or before March 25, 2025. Paper will be considered for Scopus indexed publication (AIP/ Web3 Conference/ Sprin
Physics / IEEE Computer Society) in compliance with editorial policies after the presentation in conference.

Payment Details:

Account Number: 500101012509147 Bank Name: City Union Bank Branch: Salem Main IFSC Code: CIUB0000042 MICR Code: 636054002 Mode of Payment: NEFT/RTGS
Registration Amount Details: Student/Scholar: ₹ 11000 (Registration: ₹1000 +Scopus Publication Charge:₹10,000) Faculty /Academia: ₹11250 (Registration: ₹1250 +Scopus Publication Charge:₹10,000) Industry: ₹12000 (Registration: ₹2000 +Scopus Publication Charge:₹10,000) **Additional Charges :** Offline Workshop (Only for interested) :₹ 500 Accommodation in Hostel : :₹ 500 per person

Please share the transaction details after making the online payment and complete your registration using the provided link: https://forms.gle/GHuT7ZHryCTXjVNfA .

We eagerly anticipate your participation and prompt submission.



Sona College of Technology (Autonomous) – Salem.

Department of Computer Science and Engineering

International Conference
on
Computational Intelligence, Security and Systems- ICCISS'2025

03.04.2025 & 04.04.2025

Review Form

Paper ID & Title:

ICCISS-110 – *Enhancing Text-to-SQL Conversion with Transparent and Optimized LLM Frameworks*

Evaluation (• where appropriate)

	Poor	Marginal	Acceptable	Good	Excellent
Originality / Solid Contribution				•	
Innovation				•	
Applicability					•
Technical Merit				•	
Organization and Writing			•		
System Evaluation				•	
Relevance to Conference					•
Plagiarism Percentage					

Recommendation to the Organizers (• where appropriate)

	Reject	Marginally Accept	Accept
Recommendation			•

Comments:

1. **Paper with less than 10% plagiarism are accepted.**
2. **Authors are asked to rework and reduce the plagiarism below 10% if it exceeds 10 % and within the limit 35%.**
3. **Papers above 35% plagiarism will be rejected.**

Instruction for the final paper (Please state in detail):

The paper is technically sound and presents an innovative approach to text-to-SQL conversion. However, a few areas require refinement to enhance clarity.

Revision to be carried out:

1. Improve clarity by simplifying complex sections and defining key terms for a broader audience.
2. Provide a comparative table comparing the proposed framework to existing text-to-SQL models.
3. Discuss dataset limitations, particularly its diversity and real-world generalization ability.
4. Explain the role of reinforcement learning (TRL) in performance improvement with supporting evidence.
5. Expand on future work by outlining concrete steps for multilingual support and computational efficiency.
6. Fix minor grammatical issues and improve sentence structure for better readability.