

An Interpretable, Two-Stage Text-to-SQL System

Arnav Rokde

Arizona State University

Tempe, Arizona

arokde@asu.edu

Harish Balaji

Arizona State University

Tempe, Arizona

hbalaji4@asu.edu

Karthikeyan Sivasubramanian

Arizona State University

Tempe, Arizona

ksivasu2@asu.edu

Manikandan Sundararaman

Arizona State University

Tempe, Arizona

msunda17@asu.edu

Srinitya Kondapally

Arizona State University

Tempe, Arizona

skonda29@asu.edu

Sudhersan Srinivasan

Kunnavakkam Vinchimoor

Arizona State University

Tempe, Arizona

skunnav2@asu.edu

Abstract—The task of translating natural language questions into executable SQL queries (Text-to-SQL) is crucial for democratizing data access for non-technical users. However, existing end-to-end deep learning models often function as "black boxes," making their reasoning difficult to interpret, audit, and trust, especially for complex queries. Furthermore, these systems are almost exclusively limited to read-only operations due to the significant risks of data corruption from model hallucinations. This project proposes the design and implementation of a novel, interpretable Text-to-SQL system based on a two-stage compilation process. First, we translate a natural language query, augmented with context from a Knowledge Graph (KG), into a declarative intermediate representation: Tuple Relational Calculus (TRC). This TRC form provides an auditable logical "proof" of the system's understanding. Second, a deterministic compiler safely translates the TRC into a final SQL query. The primary implementation will be a robust read-only system. As a unique research contribution, we will extend this architecture to support write operations ('INSERT', 'UPDATE') by introducing a suite of safety mechanisms, pioneering a path toward true natural language database assistants.

Index Terms—Text-to-SQL, Natural Language Processing, Knowledge Graph, Tuple Relational Calculus, Interpretability, Database Systems, Large Language Models (LLMs), Safe AI.

I. PROBLEM DEFINITION

The fundamental goal of a Text-to-SQL system is to enable users to interact with relational databases using natural language. While state-of-the-art systems have achieved high accuracy, they face two critical challenges that hinder their adoption in mission-critical enterprise environments:

- 1) **Lack of Interpretability:** Most modern systems use large, end-to-end neural networks to directly map text to SQL. This process is opaque. When the system generates an incorrect query, for example, performing an 'INNER JOIN' instead of a 'LEFT JOIN' due to a subtle linguistic nuance, it is nearly impossible to diagnose why the error occurred. This "black box" nature makes the system hard to debug, trust for critical reports, or refine without extensive retraining.
- 2) **Inability to Handle Write Operations:** The risk associated with Large Language Model (LLM) "hallucinations" makes direct generation of data-modifying language (DML) like 'UPDATE' or 'DELETE' prohibitively dangerous. A single misinterpreted request, such as "remove old records" being translated as 'DELETE FROM sales,' instead of a query with a specific date condition, could lead to irreversible data loss.

Our project directly addresses these issues by adhering to the design principle: **(TEXT + KG) → Declarative TRC → SQL (safely)**. This mirrors the classic compiler design of translating a high-level language into a logical intermediate form before generating low-level code [1], a process which guarantees interpretability and safety at each stage.

A. Primary Goal: A Read-Only Interpretable System

The main objective is to build a robust system that translates natural language questions into 'SELECT' queries. The core innovation is the intermediate generation of Tuple Relational Calculus (TRC). TRC is a formal, declarative language expressing the logic of a query (the "what") without specifying the execution plan

(the "how") [2]. By having the LLM generate TRC, we gain an explainable and auditable reasoning step that can be verified by both developers and power-users before the final, deterministic compilation to SQL. This intermediate step serves as a programmable "proof" of the system's reasoning.

B. Secondary Goal: A Prototype for Safe Write Operations

To explore the boundaries of Text-to-SQL, we will design a safety-first framework to support write operations. This is not about letting an LLM generate arbitrary DML. Instead, we will build a safety-first framework around the LLM, treating it as an assistant that drafts modifications, which are then strictly controlled and validated through a series of architectural safeguards:

- Strict intent classification to separate read from write requests.
- Constraining all DML operations to pre-defined, updatable database views, preventing direct table access.
- Enforcing a soft-delete policy, translating delete intents to non-destructive updates (e.g., 'SET is_active = false').
- Implementing a mandatory user validation step that explains the impact of an operation (e.g., "This will update 15 rows.") before execution.

II. DATA SETS

To train and evaluate our system, we will leverage established, large-scale Text-to-SQL benchmark datasets. These datasets provide diverse and complex database schemas along with a wide range of natural language questions and their corresponding ground-truth SQL queries.

- **Spider** [3]: The de-facto standard benchmark, containing 10,181 questions and 5,693 unique complex SQL queries across 200 databases with multiple tables. It is ideal for training a model that can generalize to unseen schemas and is the primary target for our read-only system.
- **SParC** [4]: An extension of Spider that focuses on conversational context, where questions are asked in sequence. This dataset is crucial for testing the system's ability to maintain state and resolve co-references in follow-up queries.
- **CosQL** [5]: A large-scale conversational dataset that includes user-in-the-loop corrections. It is valuable for exploring more interactive system designs

and potentially for training the intent classification module for our write-enabled prototype.

For our project, the target SQL queries in these datasets will be programmatically converted into their equivalent TRC representations. This transformation will create a new, structured training corpus specifically designed for our Text-to-TRC model, forming a critical part of our data preparation pipeline.

III. STATE-OF-THE-ART METHODS & ALGORITHMS

Our approach integrates classical algorithms for structure with modern deep learning techniques for semantic understanding and generation.

A. Baseline Classical DM/ML Algorithms

- **Schema Linking with Fuzzy Matching:** Before employing deep learning models, we can establish a strong baseline for identifying which database columns, tables, and values are mentioned in the natural language query. We will use TF-IDF to score keyword relevance and fuzzy string matching algorithms (e.g., Levenshtein distance) to handle minor misspellings or variations between the user's text and the schema names. This helps ground the LLM's attention.
- **Dependency Parsing:** We will use classical NLP parsers (e.g., from spaCy) to analyze the grammatical structure of the input question. The resulting dependency graph helps identify relationships between entities and can be used as a feature to guide the TRC generation process (e.g., identifying a verb phrase that indicates a filtering condition).

B. Modern AI/Deep Learning Algorithms

- **Knowledge Graph for Schema Representation:** The database schema, along with its semantic metadata (descriptions, synonyms, business rules), will be modeled as a Knowledge Graph (e.g., using Neo4j). This provides a rich, structured representation of not just the schema but the relationships within it, such as identifying primary-foreign key paths for joins.
- **Retrieval-Augmented Generation (RAG):** The user's query will first be used to retrieve the most relevant subgraph from our Knowledge Graph. This context, containing precise table/column names, data types, and join paths, is then serialized into text and "augmented" to the LLM's prompt. This grounds the model's generation in factual schema information, significantly reducing hallucinations.

- **Fine-Tuned LLM for Text-to-TRC Generation:** The core of our system is a state-of-the-art, Transformer-based language model. We will fine-tune a model on our Text-to-TRC corpus in addition. Model selection will consider trade-offs between size and resources, with candidates ranging from smaller, efficient models like Phi-3 Mini to larger, more capable models like Llama 3.1 8B or Flan-T5 [6], [7]. This size is often the sweet spot for academic projects, offering powerful performance without requiring server-grade GPUs. Furthermore, the fine-tuned LLMs will be systematically compared to existing state-of-the-art models through publicly available APIs, allowing us to identify the performance limits of using fine-tuned smaller language models in generating TRCs.
- **Rule-Based TRC-to-SQL Compiler:** This deterministic component is not based on machine learning. We will implement a compiler that parses the structured TRC output and applies a fixed set of formal logic rules to generate the final, syntactically correct SQL query. As TRC maps directly to relational calculus, which forms the theoretical basis of SQL, this translation is reliable and safe [1].

IV. RESEARCH PLAN

The project will be executed in two primary phases, focusing first on the core read-only system and then extending it with the experimental write capabilities.

A. Phase 1: Foundation – The Interpretable Read-Only System

1.1 Knowledge Graph Construction: Develop Python scripts/Information Schema queries (a SQL-standard schema that provides a self-describing, SQL-based API for system metadata) to automatically ingest database schemas (tables, columns, keys) from the Spider dataset into a graph database. Use a pre-trained LLM in a one-off process to enrich the graph by generating human-readable descriptions for tables and columns.

1.2 Data Transformation: Implement a robust SQL parser (e.g., using a library like `sqlparse`) to deconstruct the ground-truth SQL queries from Spider. Develop a rule-based converter to transform these parsed structures into their equivalent TRC representations, creating a parallel corpus for fine-tuning.

- 1.3 **Text-to-TRC Model Development:** Fine-tune the selected LLM (e.g., Llama 3.1 8B) on the new Text-to-TRC corpus. This involves extensive prompt engineering to find the optimal format for presenting the user question and the retrieved KG context. In addition, a state-of-the-art LLM will be employed to generate the TRCs, using the retrieved context and user text to provide a benchmark for comparison.
- 1.4 **TRC-to-SQL Compiler Implementation:** Build a parser for our serialized TRC format. Implement a compiler with a set of translation rules that map TRC constructs (e.g., quantifiers, predicates) to their corresponding SQL clauses (SELECT, FROM, WHERE, JOIN).
- 1.5 **Human-Feedback Caching Loop (HFCL):** Introduce a caching layer that stores verified (Text + Schema Context → TRC/SQL) pairs once a user approves the generated query. Future semantically similar inputs retrieve and adapt these cached logical forms to improve both intent understanding and query generation, reducing latency and cost. This serves as a lightweight, interpretable feedback loop similar to Reinforcement Learning from Human Feedback (RLHF) but without explicit gradient updates, enabling the system to evolve naturally through user validation and context reuse.
- 1.6 **Integration and Pipeline:** Combine all components into a single end-to-end Python-based application pipeline: Text → KG Retrieval → (Text + Context) → LLM → TRC → Compiler → SQL.
- 1.7 **Query Execution and Visualization Generation:** After query generation, the system extends its functionality by executing the query on the connected target database in a secure, read-only environment. The resulting dataset is fetched in a structured format (e.g., as a Pandas DataFrame) to enable systematic post-processing. The system, based on the user's prompt, creates the suitable visualization such as bar charts, pie charts, line plots, or heatmaps—based on the nature of the user's intent and the aggregation involved in the query. Finally, the generated visualization is rendered on the user interface, transforming raw SQL output into an interpretable and interactive visual insight.

B. Phase 2: Extension – Safe Write-Operation Prototype

- 2.1 **Intent Classification:** Fine-tune a smaller, efficient classification model (e.g., DistilBERT) to accurately distinguish read intents ('SELECT') from write intents ('INSERT', 'UPDATE', 'DELETE') based

on the user's query. This acts as a gateway to the write-handling logic.

2.2 Safe DML Framework:

- For a sample database, manually define a set of updatable views that expose limited, non-sensitive data.
- Augment the Text-to-TRC model to generate logical forms that correspond to DML operations on these specific views.
- Implement the soft-delete policy within the TRC-to-SQL compiler, ensuring any detected ‘DELETE’ intent is translated into an ‘UPDATE’ statement.

2.3 Validation and Transaction Layer:

Create a simple web-based UI component (e.g., using Flask) that presents any generated DML statement, along with a prediction of its impact, for explicit user confirmation. Ensure all executed DML is wrapped in a database transaction (‘BEGIN’, ‘COMMIT’).

V. EVALUATION PLAN

We will conduct a rigorous, multi-faceted evaluation of both the primary and secondary systems.

A. Read-Only System Evaluation

The read-only system will be evaluated on a held-out test set from the Spider benchmark.

- **Execution Accuracy:** This is the primary metric. The generated SQL query is executed on the database, and the resulting set of tuples is compared to the result from the ground-truth query. This measures the functional correctness of the entire pipeline.
- **Exact Set Match Accuracy:** A stricter metric where the structure of the generated query must match the ground-truth query, ignoring only minor variations in value or alias naming, as per the official Spider evaluation script.
- **TRC Generation Accuracy:** We will manually evaluate a statistically significant sample of the generated TRC expressions to measure the accuracy of the LLM component in isolation. This will help us perform error analysis and pinpoint sources of failure (e.g., misunderstanding of text vs. incorrect schema grounding).

B. Write-Enabled System Evaluation

Evaluation for the write-enabled prototype will focus on safety and usability, as accuracy is secondary to preventing harm.

- **Safety Constraint Adherence:** We will create a custom test suite of natural language commands designed to tempt the system into unsafe behavior (e.g., “delete all users,” “drop the sales table,” “update salaries for everyone”). We will measure the system’s success rate at correctly identifying the intent and either refusing the request or channeling it through the proper safe-handling protocol (e.g., soft-delete).
- **User Confirmation Clarity:** We will conduct a small qualitative user study (N=5) to assess whether the validation interface provides a clear and accurate understanding of a proposed write operation’s consequences. Participants will be asked to rate their confidence in approving or denying a generated DML command.

VI. PROJECT TIMELINE

The project timeline is presented in Table I.

TABLE I: Project Timeline

Weeks	Task	Description
1	KG, Data Prep & Setup	Ingest schemas, convert data, set up environment.
2	Text-to-TRC Model	Fine-tune LLM on TRC corpus with RAG.
3	TRC-to-SQL Compiler	Implement deterministic TRC parser & generator.
4	Integration & Mid-Eval	Combine pipeline, run read-only eval, mid-term report.
5	Write System Proto.	Develop classifier, safety framework, & UI.
6	Final Eval & Report	Evaluate all components, focus on safety tests.
7	Final Presentation	Prepare and deliver final project presentation.

VII. DIVISION OF WORK

The primary responsibilities for the core read-only system are divided equally among the team to ensure a balanced workload. The optional write-system extension is an additional research task led by one member.

REFERENCES

- [1] C. Draxler, “A Powerful Prolog to SQL Compiler,” CIS, Ludwig-Maximilians-Universität München, Technical Report, Aug. 1993.
- [2] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377-387, 1970.
- [3] T. Yu, et al., “Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Text-to-SQL Task,” in *Proc. EMNLP*, 2018.

TABLE II: Division of Work

Group Member	Primary Responsibilities
Manikandan Sundararaman	Knowledge Graph Architecture. <i>Additional: Safe Write-Operation Prototype Lead.</i>
Srinitya Kondapally	Data Preparation & Transformation Pipeline.
Harish Balaji	Baseline Algorithms, Schema Linking & Visualization generator.
Sudhersan Srinivasan K. V.	LLM Fine-Tuning & RAG Implementation.
Karthikeyan Sivasubramanian	Text to TRC modeling and TRC-to-SQL Compiler Development.
Arnav Rokde	System Integration, Evaluation, & Reporting.

- [4] T. Yu, et al., "SParC: Cross-Domain Semantic Parsing in Context," in *Proc. ACL*, 2019.
- [5] T. Yu, et al., "CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases," in *Proc. EMNLP*, 2019.
- [6] H. W. Chung, et al., "Scaling instruction-finetuned language models," *arXiv preprint arXiv:2210.11416*, 2022.
- [7] J. Lee, "flan-t-text2sql-with-schema-v2," Hugging Face, 2023. [Online]. Available: <https://huggingface.co/juierror/flan-t5-text2sql-with-schema-v2>
- [8] "Text-to-SQL Topic," GitHub. [Online]. Available: <https://github.com/topics/text-to-sql>
- [9] "Text to SQL Leaderboard," Papers with Code. [Online]. Available: <https://paperswithcode.com/task/text-to-sql>