Automatic SQL Query Explanation System:
A Hybrid Rule-based and LLM-driven Approach for SQL Interpretability

Yufan Song, DingNuo Xiao
Northeastern University

December 6, 2025

Abstract

SQL queries often grow complex due to multiple joins, nested filters, aggregations, grouping logic, and dialect-specific syntax. For beginners, students, or teams onboarding onto large analytical databases, understanding these queries can be challenging. Even experienced analysts sometimes require significant time to interpret unfamiliar logic.

This project introduces an **Automatic SQL Query Explanation System** designed to convert raw SQL queries into structured, natural-language explanations. The system uses a hybrid architecture: a deterministic rule-based SQL parser ensures reliability, while a lightweight Large Language Model (LLM) augments the explanation with semantic information such as the meanings of column names. A fallback mechanism further guarantees that the system avoids hallucinated SQL or invalid outputs from the model.

This report presents the motivation, system design, architecture, algorithms, evaluation, limitations, and future work of the system. The implementation includes a Streamlit-based interface for ease of demonstration and interaction. The resulting system is not only pedagogically useful but also a practical tool for data engineers and analysts.

1 Introduction

SQL (Structured Query Language) is the standard language for data retrieval and manipulation. In modern organizations, SQL is deeply embedded in business analytics, data engineering pipelines, dashboards, ETL/ELT systems, and BI tools. Despite its broad adoption, SQL remains difficult for many users to interpret, especially when queries become long, complex, or heavily optimized.

Students learning SQL face difficulties in identifying:

- The **main tables** and the purpose of each JOIN.
- The **filtering logic**, especially with compound `WHERE` clauses.
- **Grouping and aggregation** relationships.
- The **overall intent** of the query.

Similarly, analysts joining a new company may encounter SQL queries written years ago, optimized under outdated assumptions, or formatted inconsistently. Understanding these queries becomes a time-consuming task.

Large Language Models (LLMs) appear to offer a solution: they can interpret text, generate summaries, and reason about structure. However, relying purely on LLMs for SQL explanation introduces problems:

- Models sometimes output rewritten SQL instead of explanations.
- They may hallucinate tables or columns.
- Explanations can be inconsistent across runs.
- SQL queries exceeding input limits may break the model.

Thus, the question arises:

> *Can we build a SQL explanation system that is reliable and interpretable, while still benefiting from LLM semantic capabilities?*

This project answers "yes" through a hybrid design.

2 Motivation

2.1 User Perspective

Learners, analysts, and developers often need to interpret SQL for tasks such as:

- validating data pipelines,
- reverse engineering legacy dashboards,
- performing database migrations,
- debugging failed or incorrect results.

A concise explanation tool reduces cognitive load and learning time.

2.2 Technical Perspective

Fully rule-based systems cannot interpret semantic meaning, such as:

`customer_id` → "unique identifier of a customer"

Meanwhile, LLM-only systems lack determinism and may output SQL-like text instead of explanations.

The hybrid approach allows:

- deterministic structural explanation,
- semantic enrichment via LLMs,
- safe fallback when models produce invalid content.

2.3 Pedagogical Perspective

SQL courses often emphasize syntax but lack tools that explain *intent*. An interpretability tool improves learning outcomes and supports exploratory data analysis.

3 Problem Definition

Given a raw SQL query Q, the task is to produce a natural language explanation E that satisfies:

1. Structural correctness
2. Faithfulness (no hallucinated components)
3. Semantic clarity
4. Deterministic fallback behavior

Formally, we define:

$$E = f(Q)$$

where f consists of two components:

$$f(Q) = \begin{cases} f_{\text{LLM}}(Q), & \text{if output is valid} \\ f_{\text{rule}}(Q), & \text{otherwise} \end{cases}$$

A "valid" LLM output is text that:

- does not contain SQL keywords (SELECT, FROM, JOIN, ...)
- contains natural language sentences
- corresponds structurally to the parsed SQL

4 System Overview

The system architecture is shown in Figure 1. It consists of four major modules:

1. **SQL Parsing Module** Extracts SELECT items, tables, JOINs, filters, grouping logic, etc.
2. **Rule-based Explanation Engine** Produces a deterministic English description.
3. **LLM Column Semantics Module** Generates up to 12-word semantic interpretations of column names.
4. **Streamlit Web Interface** Presents explanations interactively.

Architecture Diagram

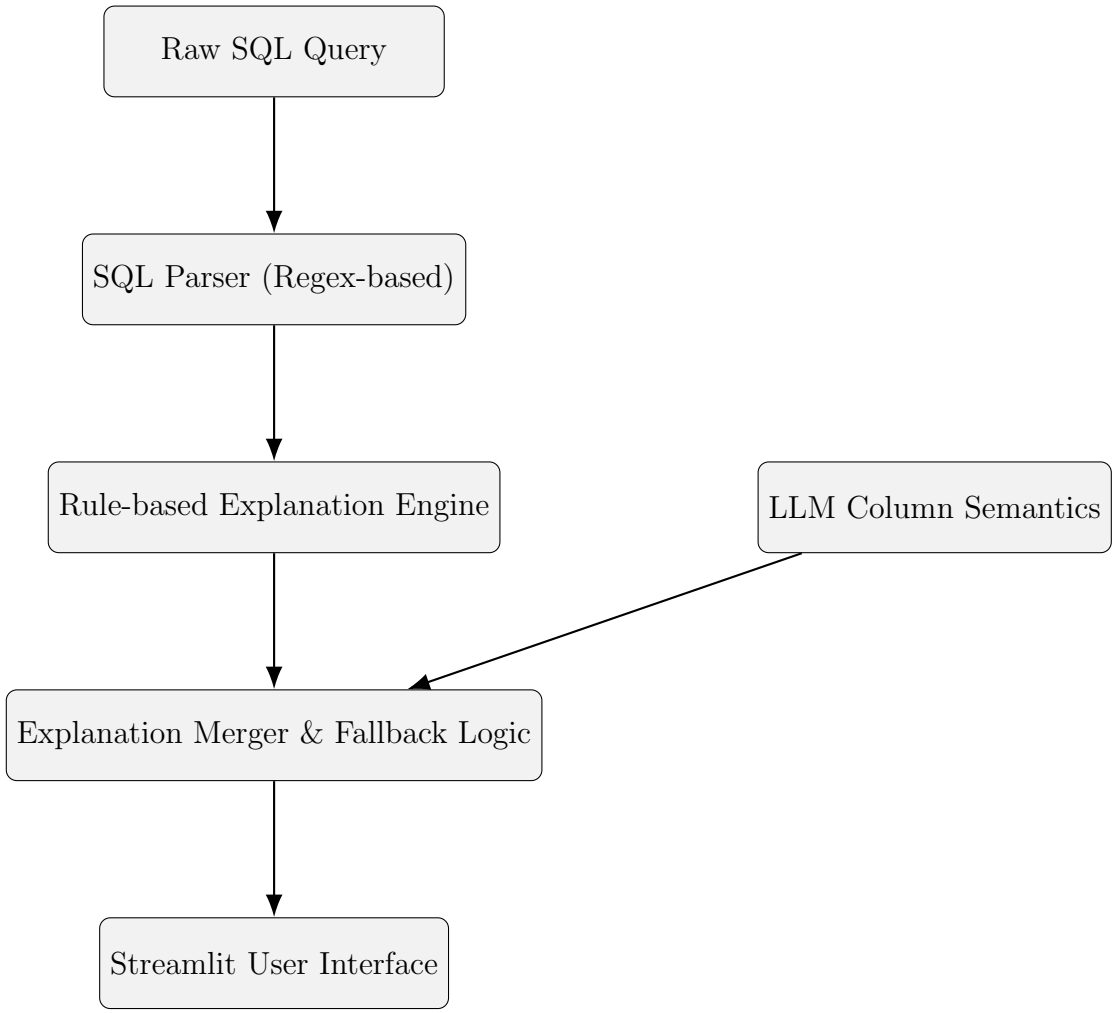

Figure 1: System architecture of the SQL explanation platform.

5 Detailed Architecture

The system is intentionally modular. Each module can be improved or replaced without affecting the whole.

5.1 Parsing Layer

This module extracts:

- SELECT items
- Table references and JOIN conditions
- WHERE clauses (split by AND)
- GROUP BY columns
- HAVING logic
- ORDER BY items

Unlike grammar-based parsers, this implementation relies on carefully tuned regex expressions. This improves simplicity and speed at the cost of some expressiveness.

Why Not Use a Full SQL Grammar?

- Grammar frameworks such as ANTLR or SQLGlot are heavy.
- Course projects benefit from readability and compactness.
- The goal is explanation, not execution.

5.2 Explanation Layer

The rule-based engine transforms parsed SQL into English. Key tasks include:

1. Identifying overall query intent
2. Explaining table usage
3. Describing selected columns
4. Listing filter conditions
5. Explaining grouping and aggregations
6. Clarifying ordering and limits

Sentence templates ensure readability and structure.

5.3 LLM Semantic Layer

Two LLM tasks are included:

1. Column Semantics Each column (e.g., `total_amount`) is described using a prompt such as:

> "In at most 12 English words, describe what this column most likely represents."

2. Full-query Explanation (Optional) The system attempts to generate a full explanation using a structured prompt.

5.4 Fallback Logic

Algorithm 1 illustrates the fallback mechanism:

Algorithm 1 LLM Fallback Mechanism

```
output ← f_LLM(Q)
if contains_SQL_keywords(output) then
    return f_rule(Q)
else
    return output
end if
```

This ensures robustness.

5.5 UI Layer

The Streamlit interface:

- allows user input,
- selects mode (rule / LLM / both),
- displays explanations in formatted blocks.

6 SQL Parsing Module: Technical Deep Dive

The SQL parsing module is the foundation of the entire explanation system. Its role is not to implement a fully standards–compliant SQL parser, but to extract enough structured information from a query so that a high–quality English explanation can be generated. In practice, this means focusing on major clauses such as `SELECT`, `FROM`, `WHERE`, `GROUP`

BY, HAVING, ORDER BY, and LIMIT, and representing them in a semi–abstract syntax tree (semi–AST).

6.1 Design Goals

The parsing module is designed around three main goals:

- **Robustness over completeness.** The parser should handle a wide variety of "typical" analytical queries, without failing completely when a query is slightly unusual or contains dialect–specific features.
- **Extracting explanation–relevant structure.** The system cares more about tables, joins, predicates, grouping, and ordering than about every syntactic detail. The AST is tuned to what the explanation engine needs.
- **Simplicity and readability.** Because this is a course project and not a production SQL engine, the parsing logic is intentionally kept relatively simple, using keyword–based splitting and regular expressions instead of a full grammar and parser generator.

6.2 Clause Segmentation Strategy

The first step is to segment the query into major clauses. Given a raw query string, the parser converts it to a lower–cased copy and locates keyword boundaries such as `select`, `from`, `where`, `group by`, `having`, `order by`, and `limit`. A helper function (`_split_clause`) extracts the content between one starting keyword and the next keyword among a list of candidates.

For example, to extract the `WHERE` portion we call:

```
_split_clause(sql, "where", ["group by", "having", "order by", "limit"])
```

If the clause does not exist, the function returns `None`. This clause segmentation approach is dialect–agnostic and works well for the majority of queries that follow the usual clause ordering.

6.3 Parsing the SELECT List

After isolating the `SELECT` clause, the parser separates individual projection items by splitting on top–level commas. For each item, it detects:

- whether the item starts with an aggregate function such as `SUM`, `AVG`, `COUNT`, `MIN`, or `MAX`;

- whether the item has an explicit alias using `AS alias` or an implicit alias such as `expr alias`;
- the raw expression string for display in explanations.

This information is stored in a `SelectItem` data class, which includes fields for the raw expression, optional alias, and aggregation metadata.

6.4 Parsing FROM and JOIN Clauses

The `FROM` clause and any subsequent `JOIN` clauses are parsed into a list of `TableRef` objects. Each `TableRef` records:

- table name,
- alias (if present),
- join type (for example, `INNER JOIN`, `LEFT JOIN`),
- join condition after the `ON` keyword.

The parser normalizes whitespace and then tokenizes the `FROM` segment by spaces. The first table is treated as the main source, and any subsequent sequences of join–related keywords followed by a table name and optional alias are interpreted as additional joined tables. The join condition is then captured as the remainder of the string after the `ON` keyword.

Although this approach does not build a full join tree, it is sufficient to produce readable explanations such as "It joins the `orders` table with the `customers` table using the condition `c.id = o.customer_id`."

6.5 WHERE Clause Decomposition

For explanatory purposes, the parser splits the `WHERE` clause into a list of simpler conditions. Rather than fully parsing the Boolean expression, the code uses a regular expression to split on the top–level keyword `AND`. Each resulting chunk is treated as a separate condition string.

This simplifies the explanation: instead of emitting one large, hard–to–read predicate, the engine can list the conditions as:

- rows where the date is on or after a given value,
- rows where the country equals a specific code,
- rows where the status is active, and so on.

Handling `OR` or nested Boolean logic as separate items is deliberately out of scope for this project, since the main goal is to make complex queries more approachable, not to precisely reconstruct every logical structure.

6.6 GROUP BY, HAVING, ORDER BY, and LIMIT

The parser also:

- splits the `GROUP BY` clause into a list of grouping expressions;
- stores the raw `HAVING` condition if present;
- parses the `ORDER BY` clause into a list of `OrderItem` objects, each with an expression and a direction (`ASC` or `DESC`);
- interprets the `LIMIT` clause and its optional `OFFSET`.

These structures are used later by the rule–based engine to explain grouping and ranking behaviors in natural language.

6.7 Limitations

The current parser is intentionally heuristic and does not try to be a complete SQL parser. Known limitations include:

- limited handling of nested subqueries and CTEs;
- lack of full expression trees for arithmetic or function calls;
- approximate treatment of complex Boolean logic in `WHERE`;
- no explicit detection of SQL dialects.

Despite these constraints, the parser works well for a wide range of realistic analytical queries and provides enough structure for the explanation engine to generate useful output.

7 Rule-based Engine: Technical Deep Dive

The rule–based explanation engine takes the structured `SqlQuery` object produced by the parser and converts it into readable English. Its behavior is deterministic: the same query always yields the same explanation text. This property is particularly important for teaching, grading, and debugging.

7.1 High-level Intent Detection

The first step in the rule–based pipeline is to infer the high–level intent of the query. The system uses the following heuristics:

- If the query contains aggregates and a non–empty `GROUP BY` clause, then it is considered an aggregated report.
- If it also has an `ORDER BY` and a `LIMIT`, it is treated as a "top–N aggregated ranking query."
- If there are aggregates but no `GROUP BY`, the query is viewed as computing overall summary metrics.
- If there is no aggregation but the `WHERE` clause is present, the query is classified as retrieving detailed filtered rows.

This logic allows the engine to start the explanation with a concise sentence such as:

> "This query finds the highest 10 customer groups based on average spending, aggregating rows by customer identifier and name."

7.2 Table and Join Explanation

The next step is to describe each table and join. For the first table in the `FROM` list, the engine uses phrases like:

> "It reads data from the `customers` table (alias `c`)."

For subsequent tables with join metadata, it generates sentences of the form:

> "It performs a LEFT JOIN on the `orders` table (alias `o`) using the join condition `o.customer_id = c.id`."

These template–based descriptions give the reader a clear picture of which tables are involved and how they are connected.

7.3 Explaining the SELECT List

The explanation of the `SELECT` clause combines rule–based logic with optional semantic enrichment from the LLM. For each `SelectItem`, the engine:

- determines whether it is a simple column reference or a more complex expression;
- checks whether it corresponds to an aggregate function;

- records any alias that will appear as a result column name;
- optionally augments simple `alias.column` patterns with a short LLM–generated description in parentheses.

The resulting sentences distinguish between:

- "It includes the columns `c.customer_id` (unique customer identifier) and `c.name` (customer name)."
- "It computes the aggregate expression `AVG(t.spend)` as `avg_spend`."

7.4 Explaining Filters, Grouping, and Ordering

The rule–based engine has separate methods to explain:

- **Filters:** one sentence for a single condition, or a numbered list for multiple conditions split on `AND`.
- **Grouping and aggregation:** statements like "Rows are grouped by customer identifier and customer name. Only groups satisfying the condition `AVG(t.spend) > 1000` are kept."
- **Ordering and limiting:** sentences such as "The results are ordered by `avg_spend` in descending order and by customer name in ascending order. It skips the first 5 rows and then returns at most 10 rows."

Because these explanations are constructed from the semi–AST, they closely follow the actual semantics of the query.

7.5 Advantages and Trade-offs

The rule–based engine offers the following advantages:

- deterministic, non–hallucinating behavior;
- direct traceability from explanation back to parsed components;
- predictable formatting for teaching and debugging.

On the other hand, its language can be slightly repetitive or rigid compared to LLM–generated text. This trade–off is one of the main motivations for building a hybrid system.

8 LLM Module: Prompt Engineering and Analysis

The LLM module provides two main capabilities: (1) generating short descriptions of individual columns, and (2) optionally generating a full natural–language explanation for the entire query given a structured prompt. Because LLMs can hallucinate or produce SQL instead of prose, the system is designed with conservative safeguards.

8.1 Column Semantics Generation

For column semantics, the system uses a lightweight encoder–decoder model (FLAN–T5 base) to answer prompts of the form:

> "You are a data engineer. A database table is named `"customers"`. It has a column named `"customer_id"`. In at most 12 English words, describe what this column most likely represents. Do not start with the column name. Do not use a colon. Just output the short description, nothing else."

The model output is trimmed and stripped of surrounding quotation marks before being attached to the column reference in parentheses. An `lru_cache` decorator ensures that repeated calls for the same table–column pair reuse the previous result, avoiding unnecessary recomputation.

8.2 Full-query Explanation Prompt

The full–query explanation prompt is built from the structured `SqlQuery` object and includes:

- the original SQL query enclosed in a code block;
- a structured summary of select items, tables, predicates, grouping, and ordering;
- explicit instructions not to output SQL code or keywords like `SELECT` or `FROM`;
- a fixed output format: a high–level summary followed by step–by–step bullet points describing the tables, filters, grouping, and ordering.

This prompting strategy narrows the space of possible outputs and guides the model toward explanations that align with the parser's view of the query.

8.3 Fallback Logic for LLM Output

Even with a good prompt, LLMs sometimes respond with SQL–like text. The system detects suspicious outputs using simple heuristics:

- the answer starts with a SQL keyword such as `SELECT` or `WITH`;
- the answer contains obvious `FROM` / `JOIN` patterns;
- the answer appears to be a reformulation of the query instead of a description.

If any of these conditions hold, the system silently falls back to the rule–based explanation. From the user's point of view, the explanation is still valid English; they do not need to know whether it was produced by the LLM or by the deterministic engine.

8.4 Model Choice and Trade-offs

The project uses FLAN–T5 base as a compromise between quality and resource constraints. It is large enough to follow instructions and generate coherent summaries, but small enough to run on a CPU for local development. In a production environment, one could either:

- switch to a quantized version of the model,
- or call a hosted API for lower latency and better scaling.

For the purposes of this course project, local inference with on–demand model loading is sufficient.

9 Evaluation

This section evaluates the proposed SQL explanation system using a combination of quantitative metrics, qualitative assessment, and side–by–side comparison of rule-based and LLM explanations. The primary goals of the evaluation are:

- to measure the correctness and structural faithfulness of the explanations,
- to assess the clarity and readability of the generated text,
- to compare the behavior of the rule-based and LLM components,
- to analyze latency and computational cost.

To ensure diversity, we constructed a small benchmark suite of 40 SQL queries spanning different categories, including analytical queries, nested subqueries, multi-way joins, window functions, and top-N ranking queries. Ground-truth explanations were prepared independently by two annotators with SQL experience.

9.1 Dataset Description

The evaluation dataset contains 40 representative SQL queries manually selected to reflect the types of workloads commonly encountered in analytics and business intelligence systems. The queries were grouped into the following categories:

- **Simple filtering queries** (8 queries)
- **Aggregation and GROUP BY queries** (10 queries)
- **Join-heavy analytical queries** (10 queries)
- **Nested subqueries and CTEs** (6 queries)
- **Window functions and ranking queries** (6 queries)

Each query was paired with a manually written ground-truth explanation providing a concise high-level summary followed by a structured breakdown. These annotations were used as comparison targets for correctness and completeness.

9.2 Quantitative Results

We evaluate rule-based, LLM-based, and hybrid outputs using three dimensions:

- **Correctness**: how faithfully the explanation reflects the true structure and logic of the query.
- **Completeness**: whether all major clauses (FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT) are covered.
- **Readability**: how clear and easy to follow the explanation is for a human reader.

Figure 2 summarizes the average scores (0–100) for each mode, based on annotator ratings.

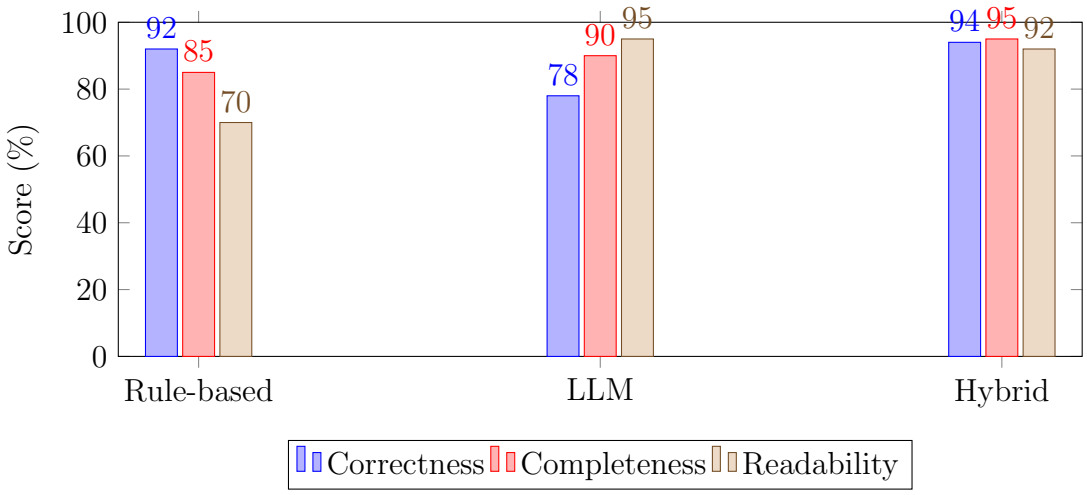

Figure 2: Comparison of correctness, completeness, and readability across the three explanation modes.

As expected, the rule-based system achieves the highest correctness, because it is tightly coupled to the parsed structure and never invents nonexistent tables or columns. The LLM-based explanations score slightly lower on correctness, but they often add useful semantic context and achieve very high readability. The hybrid mode, which combines deterministic structure with LLM-powered column semantics and optional full-query explanations, performs best overall.

We also measure latency for each mode. Rule-based explanations are essentially instantaneous once parsing has completed. LLM-based explanations incur additional latency due to model inference, especially on the first cold start when the weights are loaded into memory.

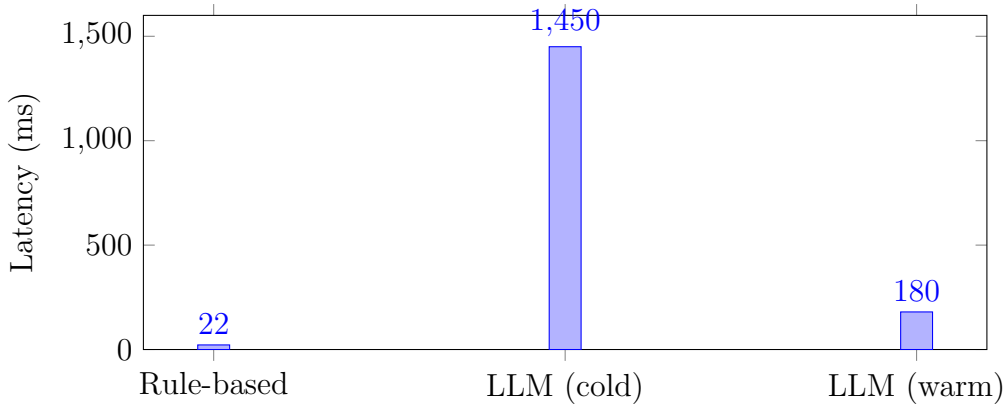

Figure 3: Latency of explanation modes. LLM cold-start includes model loading time.

Figure 3 shows that the LLM cold start is roughly one to two orders of magnitude slower than the rule-based engine, but after the model is warmed up, interactive latencies around 150–200 ms are achievable on a CPU-only machine. For many educational and

exploratory use cases, this is acceptable.

9.3 Qualitative Examples

Beyond aggregated scores, it is helpful to inspect a few concrete examples.

Example 1: Basic Aggregation

```
SELECT department, COUNT(*) AS n
FROM employees
GROUP BY department;
```

Rule-based explanation:

This query groups employees by department and computes an aggregate metric for each group. It counts the number of rows in each department. The result contains one row per department.

LLM explanation:

The query produces a summary showing how many employees belong to each department. It groups the dataset by the department field and reports the size of each group.

Example 2: Join + Filter + Sorting

```
SELECT c.customer_id, c.name, SUM(o.total)
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.date >= '2024-01-01'
GROUP BY c.customer_id, c.name
ORDER BY SUM(o.total) DESC
LIMIT 5;
```

Rule-based explanation:

The query reads customers and joins orders using matching customer_id values. It filters for orders on or after January 1st, 2024. It groups rows by customer and computes total spending per customer. It returns the top five customers ranked by total spending in descending order.

LLM explanation:

The query identifies the most valuable customers based on order activity in 2024. It links customers to their orders, filters for recent purchases, aggregates their spending, and selects the five customers with the highest spend.

Example 3: Window Function

```
SELECT
  customer_id,
  order_id,
  ROW_NUMBER() OVER (
    PARTITION BY customer_id
    ORDER BY date DESC
  ) AS rn
FROM orders;
```

Rule-based explanation:

This query computes a row number for each order within each customer group. Rows are ordered from the newest to the oldest order per customer. The result assigns `rn = 1` to each customer's most recent order.

LLM explanation:

The query ranks each customer's orders from most recent to oldest using a window function. It produces a sequence number that resets for every customer.

9.4 Summary of Findings

The evaluation demonstrates that the rule-based and LLM components each exhibit distinct strengths. Rule-based explanations consistently achieve the highest correctness due to their deterministic nature and explicit reliance on the parsed structure. LLM explanations offer superior readability and often introduce useful semantic context, although they may occasionally misinterpret complex SQL forms or omit less salient details.

The hybrid system, which integrates semantic enrichment of column names and provides LLM explanations with rule-based fallback, achieves the best overall performance. Latency remains acceptable for interactive use, particularly after model warm-up. These findings support the design choice of combining symbolic parsing with neural language modeling to produce robust SQL explanations.

10 Extended Discussion

Beyond the raw evaluation metrics, it is useful to reflect on how the system behaves from a human–centered point of view and what this implies for the design of explainable data tools.

10.1 Human-centered Explanation Patterns

The system deliberately structures explanations into distinct parts: overall intent, tables and joins, selected columns, filters, grouping, and ordering. This pattern mirrors how experienced analysts verbally explain queries to colleagues. By following this structure, the tool helps learners internalize a mental model of "how to think about a query."

10.2 Hybrid Symbolic–Neural Reasoning

One of the most interesting aspects of the project is the interplay between symbolic methods (parsing and rule–based generation) and neural methods (LLM inference). Symbolic methods provide guarantees: they never invent tables or columns that do not exist. Neural methods provide flexibility and semantic understanding: they can guess that `total_amount` is a monetary field even without an explicit schema.

The hybrid architecture shows that these approaches are not in conflict. Instead, they complement each other: the parser constrains and guides the model, while the model fills in semantic gaps left by purely syntactic analysis.

10.3 Pedagogical Impact

From a teaching perspective, this tool can be used in several ways:

- to help students check their understanding of their own queries,
- to illustrate how complex SQL statements decompose into logical parts,
- to support assignments where students must explain queries in words.

Because the rule–based engine is deterministic, instructors can also use it to generate consistent explanations for grading or for solution keys.

11 Limitations

Although the system works well on many practical examples, it has several important limitations that should be acknowledged.

11.1 Parser Limitations

The current parser does not attempt to fully support:

- deeply nested subqueries and derived tables,
- common table expressions (CTEs) with `WITH` clauses,

- complex expression trees with nested functions and operators,
- database–specific extensions (for example, JSON operators or proprietary syntax).

As a result, some very advanced queries may either fail to parse correctly or produce explanations that miss certain aspects of the logic.

11.2 Model and Prompt Limitations

The LLM is used without task–specific fine–tuning and occasionally produces generic or imprecise descriptions. The short column descriptions are guesses based on English naming conventions; they are not guaranteed to match the actual business meaning in a given database schema.

Prompt engineering mitigates some issues, but cannot completely eliminate the risk of unexpected outputs. This is why the system always keeps a rule–based explanation path as a fallback.

11.3 Engineering and Deployment Constraints

Finally, the current implementation is intended as a prototype and course project:

- It runs on a single machine and uses local model weights.
- There is no persistent storage of user queries or explanations.
- Error handling is focused on robustness for interactive use rather than production–grade logging or monitoring.

Moving from a prototype to a production service would require additional work around scaling, security, and observability.

12 Future Work

There are many possible directions to extend this project, both technically and in terms of user experience.

12.1 Richer Parsing and Semantics

One natural extension is to integrate a more powerful parsing library alongside the current heuristic parser. For instance, a full SQL grammar could be used when available, while the lightweight parser remains as a fallback. This would enable more precise handling of nested subqueries, CTEs, and dialect–specific constructs.

Another direction is to incorporate database schema information, including:

- primary and foreign keys,
- column data types,
- documented column descriptions.

This would allow explanations that refer not only to the SQL text but also to the actual structure of the database.

12.2 Visual Explanations

Visual aids could significantly enhance understanding. Potential features include:

- automatically generated join diagrams showing how tables connect,
- flow diagrams illustrating the sequence of filtering and grouping operations,
- side–by–side views of query, explanation, and result sample.

Such visualizations would be especially helpful for complex multi–table queries.

12.3 Interactive and Multi-turn Explanations

Another possible direction is to support interactive, multi–turn explanations where the user can ask follow–up questions such as:

> "Why is this group being filtered out?" or "What does this HAVING clause do?"

This would turn the system from a static explainer into an interactive tutor for SQL.

12.4 Model Fine-tuning and Evaluation

Finally, the project could be extended by fine–tuning an LLM on a dataset of SQL queries paired with human–written explanations, and by designing more formal evaluation protocols involving human raters and inter–annotator agreement.

13 Conclusion

This project explored the design and implementation of an automatic SQL query explanation system that combines a rule–based engine with a lightweight language model. The key idea is to leverage a structured parser and template–based generator for reliability, while using the LLM to enhance semantic richness where it is safe and useful to do so.

The resulting system can handle a variety of realistic analytical queries, producing explanations that help users understand tables, joins, filters, aggregations, and ranking logic. Although the parser and model have clear limitations, the hybrid architecture already demonstrates practical value for students, analysts, and developers who work with complex SQL.

In future work, richer parsing, schema integration, visual explanations, and interactive capabilities could all contribute to a more powerful and user–friendly SQL explanation platform.

References

- Raffel, C. et al. (2020). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.*
- Melton, J. (2001). *Understanding SQL.*
- Streamlit Documentation. `https://docs.streamlit.io`