# Easy SQL

Project submitted by

BIBHAS DAS

BIKRAM LOHAR

SARTHAK GHOSH

SAYAK MONDAL

AVIJIT MONDAL

MASTER OF COMPUTER APPLICATIONS (MCA)

Sister Nivedita University, Newtown

DEPARTMENT OF COMPUTER SCIENCE

Kolkata, West Bengal

2025

# EASYSQL

Project Submitted in Partial Fulfilment of the Requirements for Award of the Degree of

**Master of Computer Applications (MCA)**

**by**

**Bibhas Das, Bikram Lohar, Sarthak Ghosh, Sayak Mondal, Avijit Mondal**

**Registration numbers: 230020338214, 230020206455, 230020221612, 230020376571, 230020317583**

*Under the supervision of*

**Dr. Sayani Mondal**
Assistant Professor
Department of Computer Science
Sister Nivedita University, Newtown
Kolkata, West Bengal

DEPARTMENT OF COMPUTER SCIENCE
Sister Nivedita University, Newtown
Kolkata, West Bengal

**April, 2025**

# *Declaration*

*We hereby declare that this dissertation is the product of my/our own work, and I attest that it contains no material that resulted from collaboration, except where explicitly acknowledged in the text. Furthermore, I/we confirm that this project has not been previously submitted, either in part or in its entirety, to any other University or Institution for the purpose of obtaining any degree, diploma, or other qualification. All sources used and referenced in this dissertation are duly credited, and any borrowed ideas or information are appropriately cited in accordance with academic standards and guidelines.*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*(Bibhas Das)*
*Registration Number: 230020338214*

*Date: 30-04-2025*
*Place: SNU, WB*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*(Bikram Lohar)*
*Registration Number: 230020206455*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*(Sarthak Ghosh)*
*Registration Number: 230020221612*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*(Sayak Mondal)*
*Registration Number: 230020376571*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*(Avijit Mondal)*
*Registration Number: 230020317583*

A Satyam Roychowdhury initiative

SNU SISTER NIVEDITA UNIVERSITY

SISTER NIVEDITA UNIVERSITY

## *Certificate*

*This is to certify that the project entitled "EasySQL", submitted by Bibhas Das, Bikram Lohar, Sarthak Ghosh, Sayak Mondal, Avijit Mondal to Sister Nivedita University ,West Bengal for the award of the degree of Master of Computer Applications( MCA) is a bonafide record of the project work carried out by them under my supervision and guidance. The content of the project, in full or parts have not been submitted to any other institute or university for the award of any degree or diploma.*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*(Dr. Sayani Mondal )*
*Assistant Professor*
*Dept of Computer Science*
*Sister Nivedita University*

*Date: 30-04-2025*
*Place: SNU, WB*

# Acknowledgement

We would like to express our heartfelt gratitude to our esteemed guide, Dr. Sayani Mondal, Assistant Professor at Sister Nivedita University , WB, for her invaluable guidance, unwavering support, and encouragement throughout our project journey. Her profound knowledge, insightful suggestions, and mentorship have been instrumental in shaping our project and academic pursuits. We extend our sincere thanks to the members of the Project Committee at the Department of Computer Science , Sister Nivedita University, for their valuable feedback and constructive criticism during the evaluation of our work. Their expertise and insights have been crucial in refining the quality of our project. We would like to express our heartfelt gratitude to our HOD, esteemed Vice-Chancellor, the Class teachers and all the faculty members of our department for their unwavering support and guidance throughout our project journey. Your invaluable contributions have been instrumental in shaping this project endeavor. We are deeply indebted to our family members whose constant love, encouragement, and support have been the driving force behind our academic pursuits. Their unwavering belief in us has given us the strength to overcome challenges and achieve milestones. We also want to express our heartfelt gratitude to our friends, colleagues, juniors, and seniors for their camaraderie, encouragement, and support throughout this academic journey. Their presence and interactions have enriched us with diverse perspectives, making this journey all the more rewarding. Lastly, we extend our sincere appreciation to all those who have been our pillars of support during this project endeavor. Your encouragement, motivation, and belief in our abilities have been invaluable, and we are truly grateful for your presence in our life.

| | |
|---|---|
| Bibhas Das. | ........................................ |
| Bikram Lohar | ........................................ |
| Sarthak Ghosh | ........................................ |
| Sayak Mondal | ........................................ |
| Avijit Mondal | ........................................ |

# Contents

# List of Figures

# Abstract

EASYSQL is an interactive, command-line SQL query assistance framework designed to facilitate structured query development for novice users. The system integrates a manually curated corpus of generalized SQL patterns embedded with dynamic placeholders, which are replaced at runtime using live metadata extracted from an active MySQL database connection. Query tokenization and syntactic analysis are performed using sqlparse and ANTLR-based grammar processing to enable intelligent, context-sensitive autocompletion and structure-aware suggestions. The terminal interface, supports advanced features including multidimensional auto-complete menus, real-time syntax highlighting, and customizable key bindings to enhance user interaction. Post-query validation mechanisms are implemented for data manipulation statements, through rigorous verification of value-type alignment against the underlying schema definitions, thereby enforcing semantic integrity. Query execution results are presented in a formatted tabular structure leveraging the rich library to maximize output legibility. Furthermore, EASYSQL incorporates an adaptive learning module that monitors user interactions to dynamically reprioritize suggestion relevance based on usage frequency, progressively optimizing user experience. Unlike fully automated SQL generation systems, EASYSQL emphasizes assisted query formulation, reinforcing syntactic and semantic learning while maintaining user control. The system is engineered for high compatibility with MySQL environments and prioritizes schema-aware guidance, post-execution validation, and interactive learning methodologies within a controlled CLI-based ecosystem.

# Chapter 1

# Introduction

Data refers to raw facts, figures, or observations that can be processed to derive meaningful information. In computing, data is typically stored, retrieved, and manipulated in structured or unstructured forms. A database is an organized collection of structured data, designed to efficiently manage, update, and retrieve information. Databases serve as the backbone of modern applications, enabling systematic storage and access to large volumes of data. Structured Query Language (SQL) is the standard programming language for interacting with relational databases, which store data in tables with defined relationships. SQL acts as a bridge between users and databases, allowing them to create, modify, and query data efficiently. The connection between databases and SQL is fundamental—databases provide the storage framework, while SQL serves as the interface to communicate with and manipulate this data.

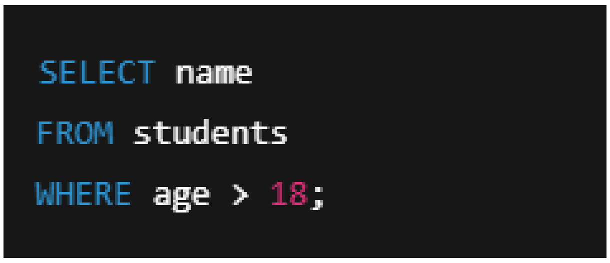
A SQL query is a command written in SQL syntax to perform operations such as retrieving, inserting, updating, or deleting data. As a query language, SQL enables users to define what data they need without specifying how to retrieve it, leaving the optimization to the database management system (DBMS). Software applications rely on SQL to interact with databases, making it an essential tool for developers, analysts, and database administrators. Despite its importance, SQL presents challenges for beginners and students. Syntax errors, difficulty in comprehending query structure, and confusion regarding database schemas often hinder effective learning. To overcome these issues, our project presents EASYSQL—an intelligent SQL suggestion tool designed for educational purposes. EASYSQL assists learners by offering real-time hints, smart suggestions, and error corrections as they compose SQL queries. By guiding users on what to type next, how to format queries, and how to avoid common mistakes, EASYSQL aims to make the SQL learning process more intuitive, interactive, and efficient. This literature survey examines early and contemporary research on SQL learning tools, including web-based systems. By analyzing existing technologies and methodologies, we identify gaps where EASYSQL can provide innovative solutions, ultimately enhancing database education for students[1] .

In traditional SQL, a set of fundamental commands is used to interact with and manipulate data within a database. Each command serves a specific purpose, enabling users to perform operations such as retrieving, inserting, updating, and deleting data. To retrieve data from a table, the `SELECT` statement is used, allowing users to query the database and retrieve the desired information based on specified conditions. Based on the preceding discussion, the following SQL query illustrates the retrieval of specific data from a database.

**It selects the names of all students whose age exceeds 18 years:**

This query retrieves the values of the columns `name` for all records in the table `students` where the attribute `age` exceeds 18 years. The WHERE clause filters records according to the specified condition, demonstrating SQL's capability for precise data extraction.

Early SQL suggestion systems like SnipSuggest[2], QueRIE[3], and SQLSUGG[4] were innovative but limited by static logic, relying heavily on historical query logs and fixed templates. While they helped beginners with context-aware suggestions, they struggled with ambiguity,



```
SELECT name
FROM students
WHERE age > 18;
```

Figure 1.1: Example Of SELECT keyword

lacked adaptability, and performed poorly on complex or large-scale queries. Later tools like DialSQL[5] and Agrawal's System[6] introduced schema-awareness and guided query generation but were still constrained by rigid rules and limited semantic understanding. Recent advancements have embraced AI-based techniques such as RNNs and Q-Learning, enabling more accurate, dynamic SQL predictions[7].Tools like SQLectron have further simplified query creation through NLP and visual interfaces, making SQL accessible even to non-experts[8]. However, modern platforms like SQL Fiddle, PopSQL, and DBeaver have improved the user experience with real-time execution, intelligent autocomplete, and collaborative features, bridging the gap between learning and professional use. Overall, the evolution reflects a shift from static systems to adaptive, intelligent environments tailored for efficiency and ease of use. As users type their own queries, EASYSQL provides context-aware,real-time suggestions based on the database schema and the partially written query. This approach promotes active learning by guiding users through each component of the query, helping them understand the structure and logic behind SQL statements, rather than simply relying on fully automated outputs. By offering intelligent assistance without removing user involvement, EASYSQL effectively bridges the gap between novice understanding and proficient SQL writing skills.

**EASYSQL** aims to assist beginner students in learning SQL by providing real-time, context-aware query suggestions and error correction. Instead of generating full queries, it guides users step-by-step, helping them understand SQL structure, reduce syntax errors, and build confidence. The tool promotes active learning by offering intelligent support based on user input and database schema, making the SQL writing process more interactive and educational. A core feature of EASYSQL is its **runtime error detection and correction module**, which actively monitors user input to identify and correct common errors such as incorrect syntax, missing clauses or misused functions. Rather than simply pointing out errors, the system offers actionable suggestions to guide the learner toward correct query formulation.

This work presents an intelligent terminal-based SQL suggestion system that retrieves real-time database schema information to provide accurate query assistance. By combining lightweight single field query storage with a stack-based merging strategy, it efficiently handles both simple and complex queries. The system accelerates SQL writing, minimizes errors, and improves the user's understanding of proper syntax. Unlike existing tools, it uniquely offers real-time schema-aware suggestions directly in the terminal environment.

## 1.1 Thesis Organization

1. Chapter 1:Introduction
   Introduces the fundamentals of SQL and databases, outlines the challenges faced by beginners in query formulation, and presents the motivation behind EASYSQL, an intelligent suggestion tool for SQL learners. It identifies gaps in existing solutions (lack of real-time guidance vs. over-automation) and summarizes the thesis objectives and contri-

butions.

2. Chapter 2:Literature Survey surveys foundational and contemporary SQL learning tools, analyzing their limitations in real-time feedback, adaptability, and educational value. Position EASYSQL as a bridge between passive IDEs and fully automated NLP systems, emphasizing its context-aware, learner-centric approach.

3. Chapter 3:Methodology EasySQL's methodology combines input tokenization, ontology-based pattern matching, and schema-aware suggestions to provide real-time SQL assistance. The system analyzes queries using a stack-based approach for subqueries and performs error correction against templates. This integrated process delivers accurate recommendations while teaching proper query construction, balancing immediate help with long-term learning.

4. Chapter 4: Result and Analysis This chapter outlines the implementation of the EASYSQL query suggestion and correction system. It details the step-by-step suggestion process, real-time input handling, and schema-based validation. Key development challenges—like handling subqueries, multiple fields, and schema matching—are discussed along with their solutions.

5. Chapter 5: Conclusion EASYSQL evolved from a basic tokenizer into a real-time SQL query assistant for beginners. By combining schema-aware placeholder replacement with dynamic suggestions, it helps users build valid queries efficiently. The system simplifies query formulation and minimizes errors through intuitive, real-time guidance.

6. Chapter 6: Future Scope Future development of EASYSQL aims to integrate lightweight ML models for smarter token prediction and ranking, apply clustering and reinforcement learning for adaptive suggestions, and enable cloud deployment for scalability and collaboration. It also envisions a full-featured web and desktop application, along with IDE extensions and enterprise features like access control and logging.

# Chapter 2

# Literature Survey

In this study, both foundational efforts and contemporary research related to SQL learning tools, including a range of web-based systems designed to support database education. By examining a variety of technologies and pedagogical methods developed over time, this study aims to capture how learners interact with SQL training platforms, what instructional strategies have been effective, and where limitations still exist. Emphasis is placed on identifying the evolution of user interfaces, levels of automation, feedback mechanisms, and adaptability to different learner needs. Through this comprehensive overview, we aim to highlight the current gaps in these systems—particularly in areas such as real-time query validation, intelligent feedback, and context-aware suggestions. These insights help pinpoint opportunities where EASYSQL can bring meaningful improvements, offering enhanced functionality, better usability, and increased educational value for students engaged in learning database systems. By aligning with these identified needs, EASYSQL is positioned to deliver a more supportive and intuitive learning experience.

Early efforts to simplify SQL querying introduced tools that were innovative for their time but limited by static logic and evolving database demands. Snip Suggest is a SQL query suggestion system that was intended to help users by offering context-sensitive suggestions from past query logs. It simplified query creation, particularly for novice SQL users, by recycling frequently written query fragments. Nonetheless, because its suggestions were taken from static past data, the system could not learn dynamically or adapt to novel query patterns. Consequently, it tended to generate inappropriate suggestions for more complicated or novel query types[2]. SQLSUGG is a query suggestion system that supports users—particularly novices—in creating SQL queries by converting straightforward keyword inputs into full SQL statements. It sought to minimize the syntactic complexity of query construction by letting users concentrate on what they were looking for, not how to syntactically construct the query. Technically, the system depended on a collection of pre-defined query templates to infer user intent. This was effective with simple queries but struggled with ambiguous input or sophisticated query structures. In addition, SQLSUGG experienced performance problems on large databases, restricting its responsiveness in more data-intensive applications[4]. QueRIE is a SQL query suggestion system intended to assist users in navigating intricate datasets—particularly in scientific contexts—by monitoring previous user activity and query logs. The application watches how users engage with the database and provides suggestions for subsequent queries based on those patterns. This helped users find new knowledge without having to write each SQL query anew. Technically, QueRIE applies log mining methods to identify commonly accessed query paths and utilizes them to create customized suggestions. Nevertheless, its performance was hampered by its heavy reliance on past experience, which usually resulted in repetitive or biased suggestions. As such, it performed poorly in servicing varied or pioneering exploration strategies deviating from accustomed user behavior[3].

DialSQL is an intelligent SQL query suggestion service that provides real-time, adaptive completions through the examination of both the database schema structure and a user's past queries. The objective was to decrease the amount of effort required to write SQL, particularly in complicated databases, by providing smart, context-sensitive recommendations based on the user's previous actions. From a technical perspective, DialSQL combines schema knowledge with past query patterns to enhance the quality of its recommendations. Yet the quality of the system depended significantly on the availability of well-structured, high-quality schemas. It also lacked deeper query optimization techniques and struggled with performance when working on large-scale or highly dynamic datasets, which impacted its usability in enterprise-level environments[5]. Agrawal's System was built as a study aid that could generate SQL queries automatically through the use of RDFS (Resource Description Framework Schema) models—basically employing structured metadata to translate database abstractions into executable queries. The concept had merit: by taking students through pre-established patterns, it mitigated the upfront drag of hand-writing SQL[6]. Nevertheless, the system's practical utility was constrained by significant limitations. It lacked deep semantic analysis — it could not truly "understand" a user's intent or adapt to nuanced query requirements. Furthermore, it struggled with complex queries, such as multi-level joins and correlated subqueries, because its rule-based approach was rigid and could not scale beyond simple query patterns.Although these early systems laid important groundwork, they are now considered outdated due to their limited flexibility, reliance on static heuristics, and inability to meet the demands of modern, dynamic SQL environments. Recent developments in Recurrent Neural Networks (RNNs) and Q-Learning (Reinforcement Learning) have improved SQL query prediction.It created a model that predicts a user's future SQL query by examining query sequences with temporal machine learning models such as RNNs and Q-Learning. Pre-computation of probable queries can enhance response times by 30-50% when the predictions are accurate. Q-Learning surpassed RNNs by using past query patterns, making 15-20% more accurate predictions and speeding up query execution by speculative processing[7].

In addition,presented an NLP-based system with visual interactions that allows users to query databases in natural language or visually (e.g., dragging tables, clicking filters). The system automatically converts these inputs into accurate SQL queries, with 92% query accuracy and allowing non-experts to construct correct queries 60% more frequently. It also permits users to construct queries 3-5x faster, with 80% fewer keystrokes, and cuts debugging time by 40% [8]. The proposed method improves text-to-SQL conversion by reusing tokens and leveraging query history. Tested on Spider, Sparc, and CoSQL, it boosts accuracy and F1-score by 8–15% over models like Seq2Seq and SQLNet, while reducing latency through incremental refinement—avoiding full-query prediction risks of speculative approaches like RNNs and Q-Learning [9]. Recent advances in RNNs and Q-Learning have enhanced SQL query prediction.It used RNNs and Q-Learning for SQL query prediction, with Q-Learning boosting accuracy by 15–20% and cutting response times by 30–50% via speculative execution—speeding up dynamic SQL generation using past query patterns [10]. SQLSketch's sketch-based decomposition makes it robust for cross-domain NL-to-SQL, while the GreatSQL dataset enables better training. Future work includes extending it for complex queries (nested subqueries, GROUP BY)[11]. This paper discusses Natural Language Interface to Databases (NLIDB), in which users query databases with natural language rather than SQL. It talks about the application of semantic parsing and neural models that are approximately 80 percent accurate in text-to-SQL translation. Current models such as SQL-Net, Syntax SQL-Net, Grammar SQL, IR-Net, Edit SQL, and RAT-SQL are contrasted against their support for aggregation, column choice, and clauses. The article also addresses speech-to-SQL models and the Debug-It-Yourself (DIY) paradigm, emphasizing practical limitations. Future enhancements such as ontological acyclic directed graphs are proposed to improve system precision[12].

It is hard for computer science students to learn SQL, which results in students failing database classes. SQLValidator was created by Otto-von-Guericke-University to assist students with exercises, quizzes, and instant feedback. This thesis enhances the feedback by associating exercises with the most suitable lecture slides using cosine similarity and tf-idf techniques. In this manner, students receive targeted advice on where to focus their study if they are having trouble with an exercise[13]. Data is important to industries, businesses, and governments, and most of it is kept in relational databases with SQL. But composing SQL queries needs special expertise and recalling complicated syntax. This paper suggests a way to create SQL queries from natural language, such as spoken (audio) input, with NLP and Deep Learning with an LSTM model. The system assists users, such as those with Repetitive Stress Injury (RSI) or minimal SQL knowledge, by generating the required SQL queries automatically [14]. Data is extremely critical for businesses, sectors, and governments. The majority of data is stored in databases, and individuals work with SQL to handle it. However, SQL can be difficult to write because you need to be aware of the rules and recall complex commands. To simplify this, this project establishes a platform where an individual can simply speak or type in plain language, and the system will generate the appropriate SQL query automatically. It employs intelligent technologies such as NLP (Natural Language Processing) and LSTM (a deep learning model type). This program is also useful for individuals with pain due to excessive typing (RSI) or for those who are unfamiliar with SQL[15]. This project introduces Semantic Synthesis, a method that translates natural English language into accurate SQL queries using Generative AI and Large Language Models (LLMs) with transformers. It aims to make database interaction accessible for users without SQL expertise by offering an intuitive interface and a transparent algorithmic breakdown of the query generation process. The system's effectiveness is tested across various domains like e-commerce, healthcare, education, and human resources, highlighting its potential to democratize database access and support data-driven decision-making[16].

SQL Fiddle is an excellent browser-based SQL sandbox that allows you to rapidly spin up database schemas and execute queries live, with a neat split-screen layout having schema setup on the left and query execution on the right - ideal for playing with SQL syntax in various database engines such as MySQL, PostgreSQL, and SQL Server without local setup. PopSQL has become a go-to collaborative SQL editor that bridges the simplicity of new note-taking apps with robust database tools, allowing teams to write, share, and visualize queries together in real-time while providing intelligent autocomplete, version control, and native charting to make data workflows seamless. DBeaver (dbeaver.io) is the database workbench of choice among developers who require a single rock-solid tool that can do everything - from coding sophisticated SQL queries with intelligent autocomplete to graphically browsing schemas across MySQL, PostgreSQL, MongoDB and dozens of other databases, and providing convenient data export capabilities with both free (community edition) and high-powered (enterprise) versions.

Existing SQL learning tools, including database IDEs and NLP-based query generators, fail to adequately support the learning process for SQL beginners. While traditional tools like MySQL Workbench provide limited error feedback only after query execution, newer NLP-based solutions that automatically convert natural language to SQL create a different problem—they bypass the learning process entirely by generating complete queries without explaining their structure. This creates a critical gap in SQL education: current approaches either offer no real-time guidance (traditional IDEs) or remove the learning opportunity altogether (NLP translators). EASYSQL addresses this gap through its innovative query suggestion methodology. Diverging from NLP-based systems that produce complete SQL statements automatically, our solution offers intelligent, real-time guidance that adapts to users' input, providing contextual hints and completions during the query construction process. This balanced approach maintains learner engagement while systematically building SQL proficiency, overcoming two key limitations of current technologies: first, it avoids creating over-reliance on automated solutions that hinder skill development, and second, it provides the immediate support that conventional tools fail to deliver. The optimal position between complete automation and minimal assistance, EASYSQL enables meaningful learning progression. The system emphasizes conceptual understanding through interactive guidance, focusing not merely on producing correct queries but on cultivating the thought process behind effective SQL writing. This educational focus meets a critical need in database instruction - fostering both practical query-writing abilities and deeper comprehension of database principles, aspects that existing tools consistently overlook in favor of either full automation or passive execution environments.

The objective of this project, EASYSQL, is to develop an intelligent, interactive SQL query suggestion system tailored to support beginner-level learners in understanding and writing SQL more effectively. EASYSQL addresses challenges such as query structuring, syntax formulation, and schema navigation by providing real-time, context-aware suggestions during query construction. The system uses a manually curated set of generalized SQL patterns, dynamically adapting placeholders with actual schema values from a connected database to offer relevant recommendations. Unlike systems that fully automate query generation, EASYSQL emphasizes incremental learning by allowing users to compose queries independently with guided assistance. The objective of this study has been divided into two broad categories:

- Query suggestion generation
- Query Correction

In the next chapter, a detailed explanation has been provided to develop the application.

# Chapter 3

# Methodology

The approach used in EasySQL aims at supporting novice-level users to construct correct SQL queries using live assistance. The system focuses on improving the SQL writing experience by providing real-time suggestions and dynamic error correction. It ensures that users develop a better understanding of SQL query structures while minimizing common mistakes.

## 3.1 Use Case Diagram of System

The Use Case Diagram Figure 3.1 captures the global interaction between the User, the EASYSQL Program, and the Database. It presents the core functions of the system, such as writing queries, creating real-time suggestions, making token lists, checking queries, correcting errors, and running queries. The diagram further illustrates system functions such as retrieving the database schema, error handling, and keeping scores. Various system components collaborate to guide the user through a structured and systematic process, ensuring that the query is correct, valid, and successfully executed, while also offering learning support and error feedback.

## 3.2 Knowledge Graph

Knowledge graphs Table: 3.1 are structured representations of information that capture entities, their attributes, and the relationships between them. They provide a powerful way to organize and query complex, interconnected data. Knowledge graphs have gained prominence in recent years due to their ability to represent and reason about information in a way that's closer to human understanding[1]. A knowledge graph Table: 3.1 works by organizing and linking data points (entities) through relationships, extracting the information about those entities and their relationships from the data itself. The information provided in a knowledge graph is usually sourced from multiple datasets. Data that comes from various sources may be structured differently, so it needs to be standardized, processed,, and unified to be used in a graph. The step often involves entity resolution — the process of identifying the records that reference the same real-world objects. Some knowledge graphs have an additional semantic layer with descriptions called ontologies, which explain exactly what the entities are and the relationships between them. The knowledge graph acts as a key component of the project by enabling a semantic and structured representation of SQL queries. Rather than viewing queries as separate text strings, the knowledge graph structures each query as interrelated elements like query type, table name, conditions, and privileges. The organization enables the system to realize inter-dependencies between various pieces of a query, such that query generation, modification, and recommendation become efficient. When a user gives a requirement, the system can navigate through the knowledge graph to find appropriate queries or build new ones by combining suitable pieces. The utilization of a knowledge graph increases flexibility, scalability, and smart query suggestion, making the system able to help users generate correct SQL queries

[1]https://www.ibm.com/think/topics/knowledge-graph

based on limited input. Also, it enhances system maintainability, as extensions or updates can be easily handled by the mere addition or modification of nodes and relationships in the graph. The following describes the step-by-step approach that has been adopted in developing fundamental functionalities such as query suggestion and error detection or correction, and also the tools and development process utilized.

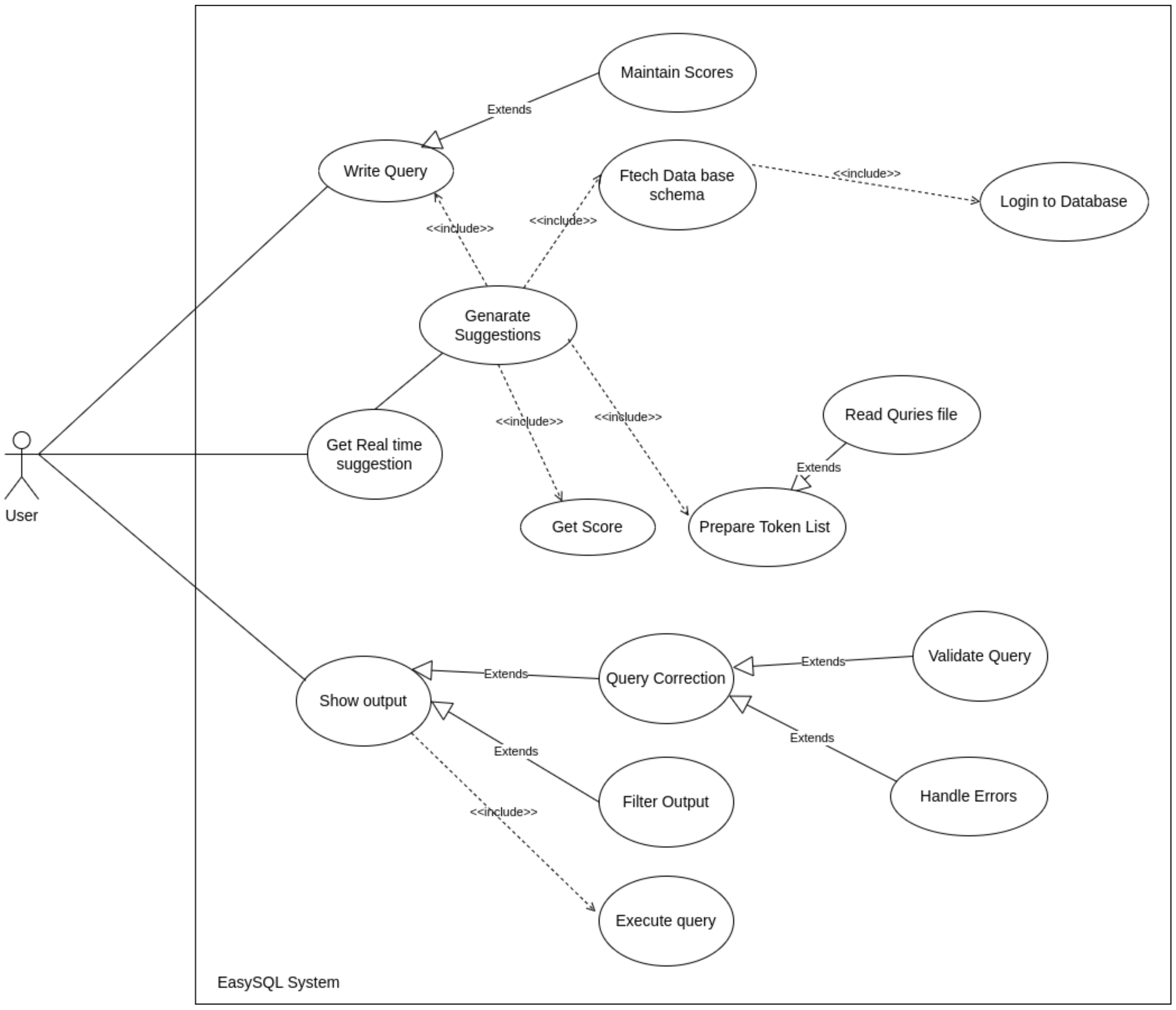


Figure 3.1: Use case diagram

### 3.2.1 Create ontology

The create ontology shown in Figure:3.2, illustrates how various database objects such as table, view, index, procedure, function, trigger, and sequence are constructed. It explains relationships such as field definitions, constraints, operation triggers, and how to create structures dynamically with as select.

### 3.2.2 Alter ontology

The alter ontology Figure : 3.3 specifies how structures of tables can be altered by adding, changing, renaming, or dropping columns and constraints. It specifies flows for modifying character sets, indexes, partitions, and handling table properties such as auto increment and comments.

### 3.2.3 Drop ontology

The drop ontology directs removal of database objects like tables, views, indexes, triggers, and databases. It stresses features such as if exists, and operations such as cascade to remove dependencies or restrict to avoid deletions by mistake.

### 3.2.4 Select ontology

The chosen ontology Figure : 3.4 explains the retrieval process from beginning with selecting fields, joining tables, filtering where, grouping by, sorting using order by, using window functions such as rank(), row number(), and combining results using union or intersect. It spans simple to very complex data querying paths.

### 3.2.5 Insert ontology

The insert ontology Figure : 3.5 translates the process of inserting into a table, naming fields and their values. It includes how other clauses such as on duplicate key update, returning, and functions applied to inserted values such as upper(), case are formatted. It emphasizes simple and complex insert operations.

### 3.2.6 Update ontology

The update ontology Figure : 3.6 links the process of assigning field values, applying conditions via where, and utilizing functions such as round(), now(), and window functions. It demonstrates how an update logically cascades into operations, expressions, conditions, and even transaction control using begin, commit, and rollback.

### 3.2.7 Delete ontology

The delete ontology Figure : 3.7 illustrates how to delete rows from a table, optionally with complex conditions in where like in, between, is null, and joining to operations like order by, limit, and transactional control. Joins and advanced options such as cascade delete are also described.

### 3.2.8 Grant ontology

The grant ontology Figure : 3.8 bridges privilege assignment to users, translating privileges such as select, insert, update, delete, and administrative rights. It illustrates how permissions are allocated to particular database objects such as tables, procedures, and schema, with control at the field level.

### 3.2.9 Revoke ontology

The revoke ontology Figure : 3.9 represents the undo of permissions that were issued through grant. It reflects the privilege structure, illustrating how access to tables, functions, sequences, and fields can be removed from users in order to preserve database security and control.

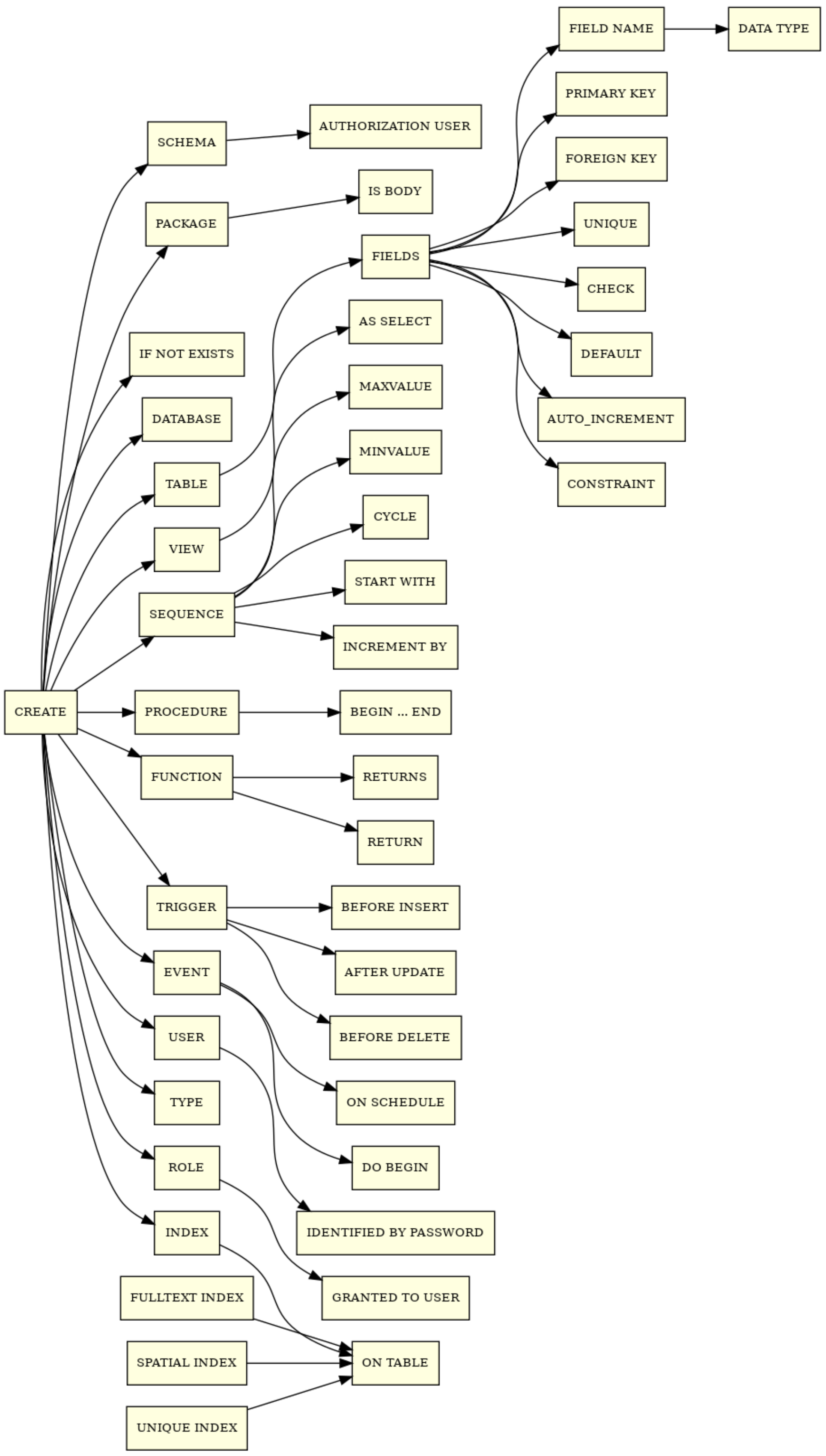


Figure 3.2: knowledge graph of Create

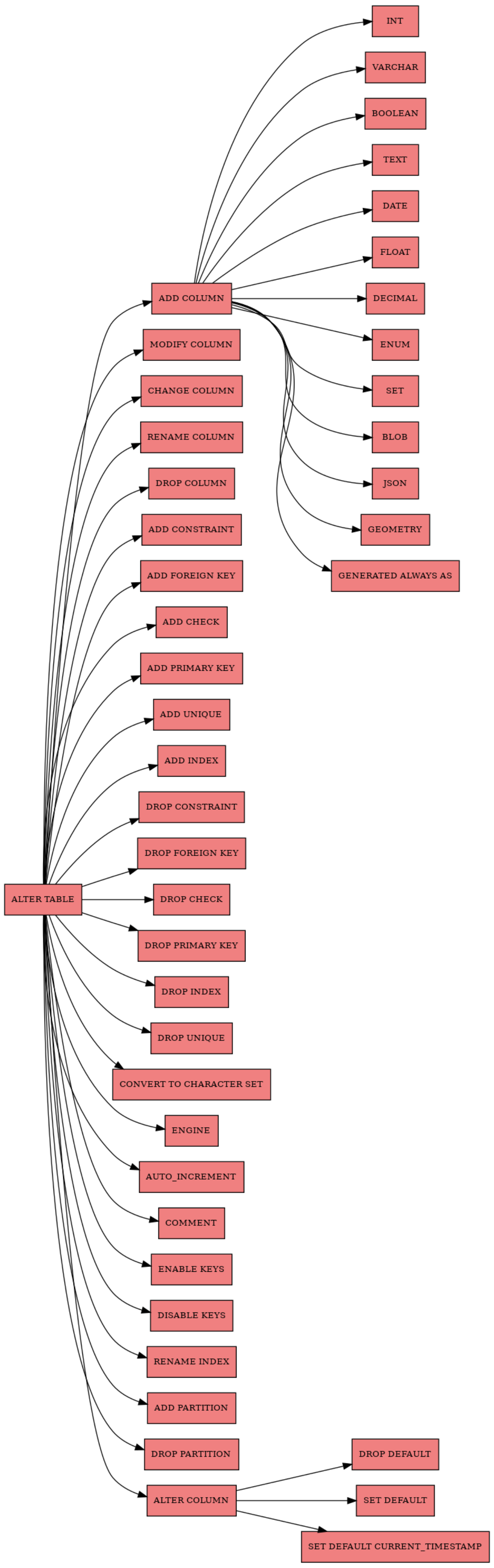


Figure 3.3: knowledge graph of Alter

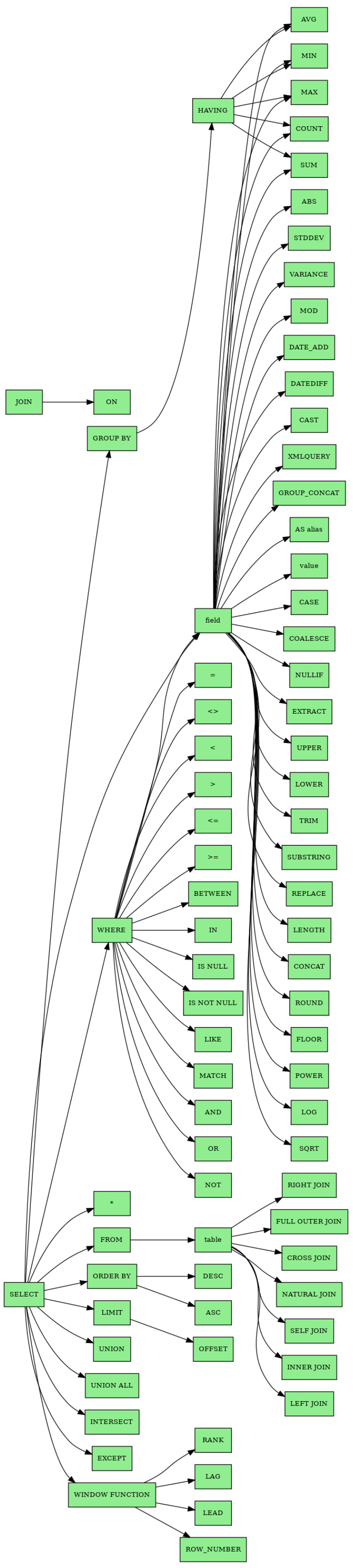


Figure 3.4: Knowledge graph of Select

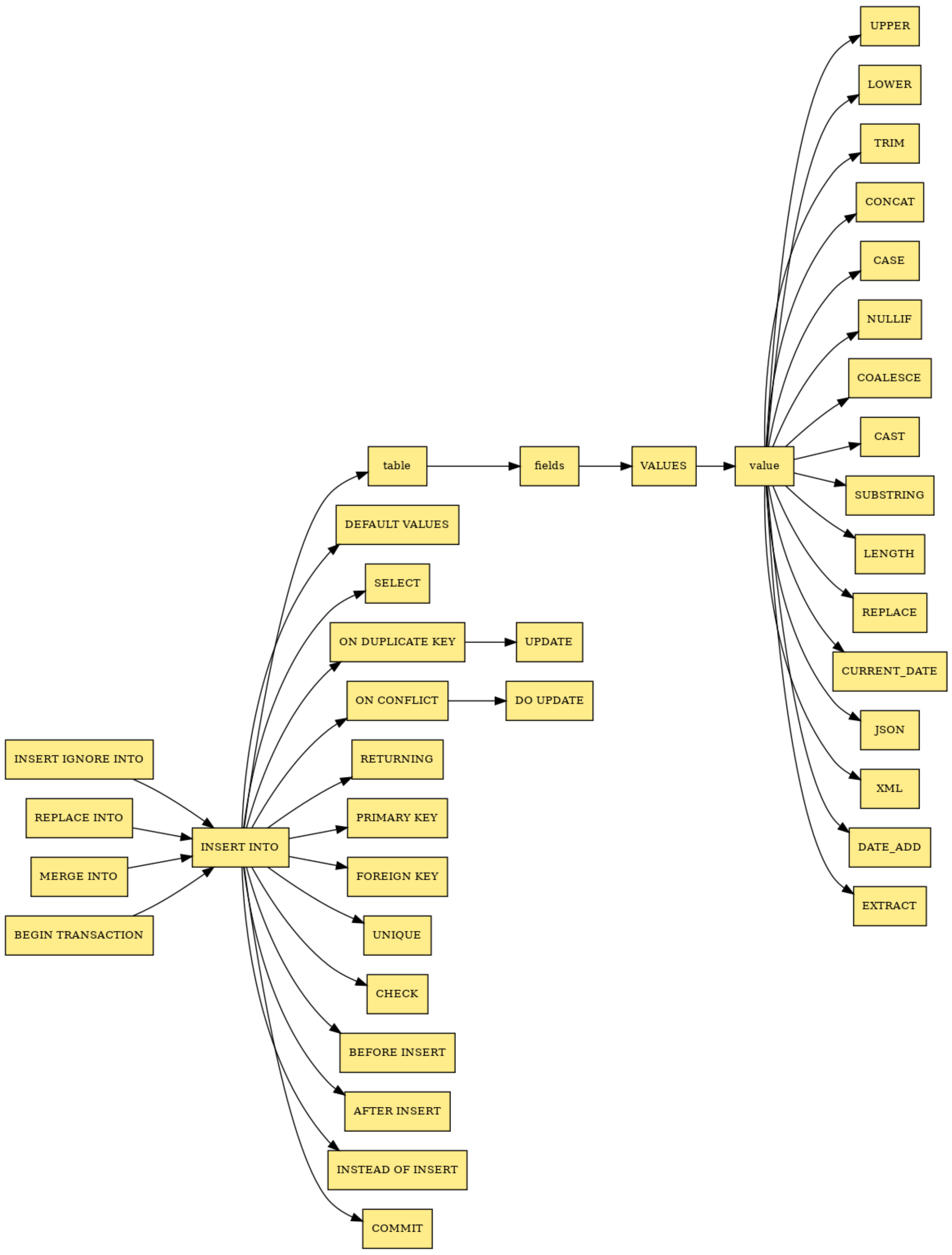


Figure 3.5: Knowledge graph of Insert

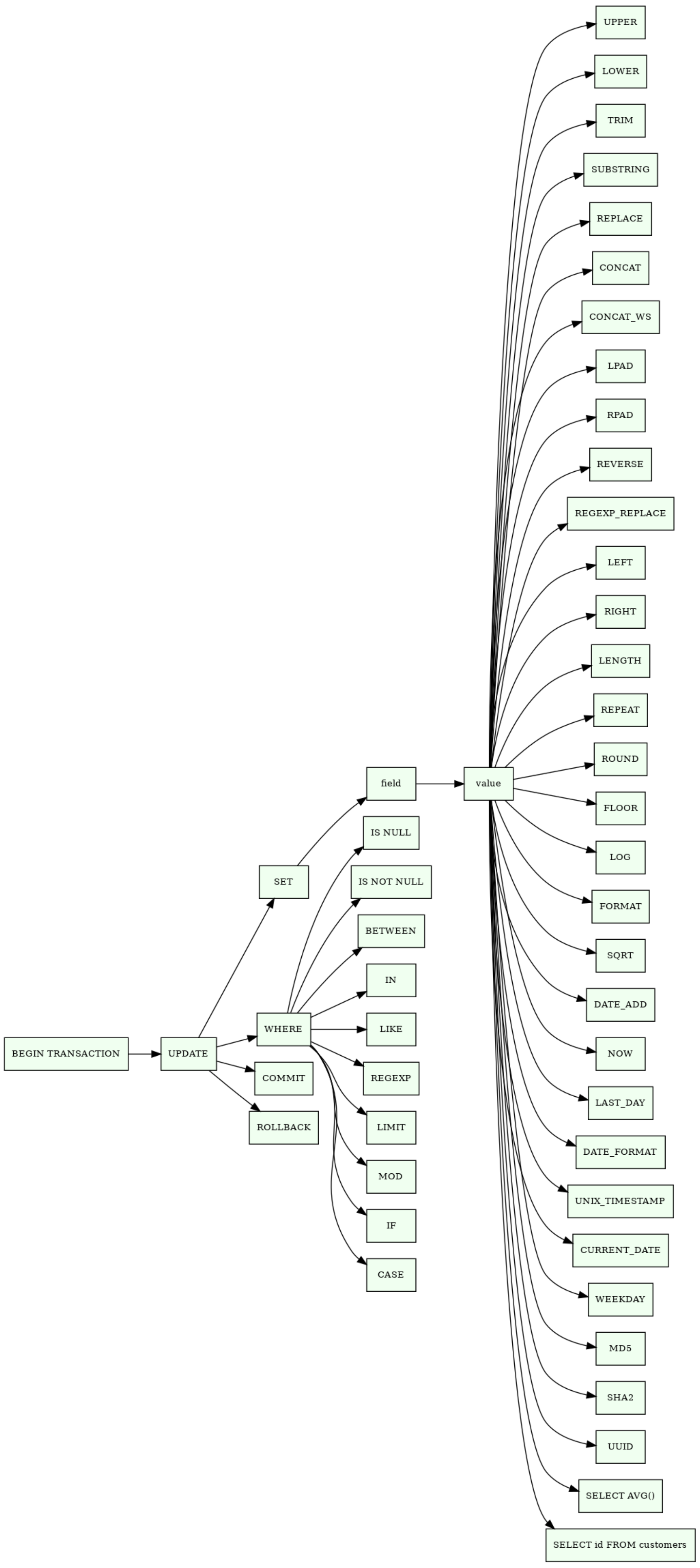


Figure 3.6: Knowledge graph of Update

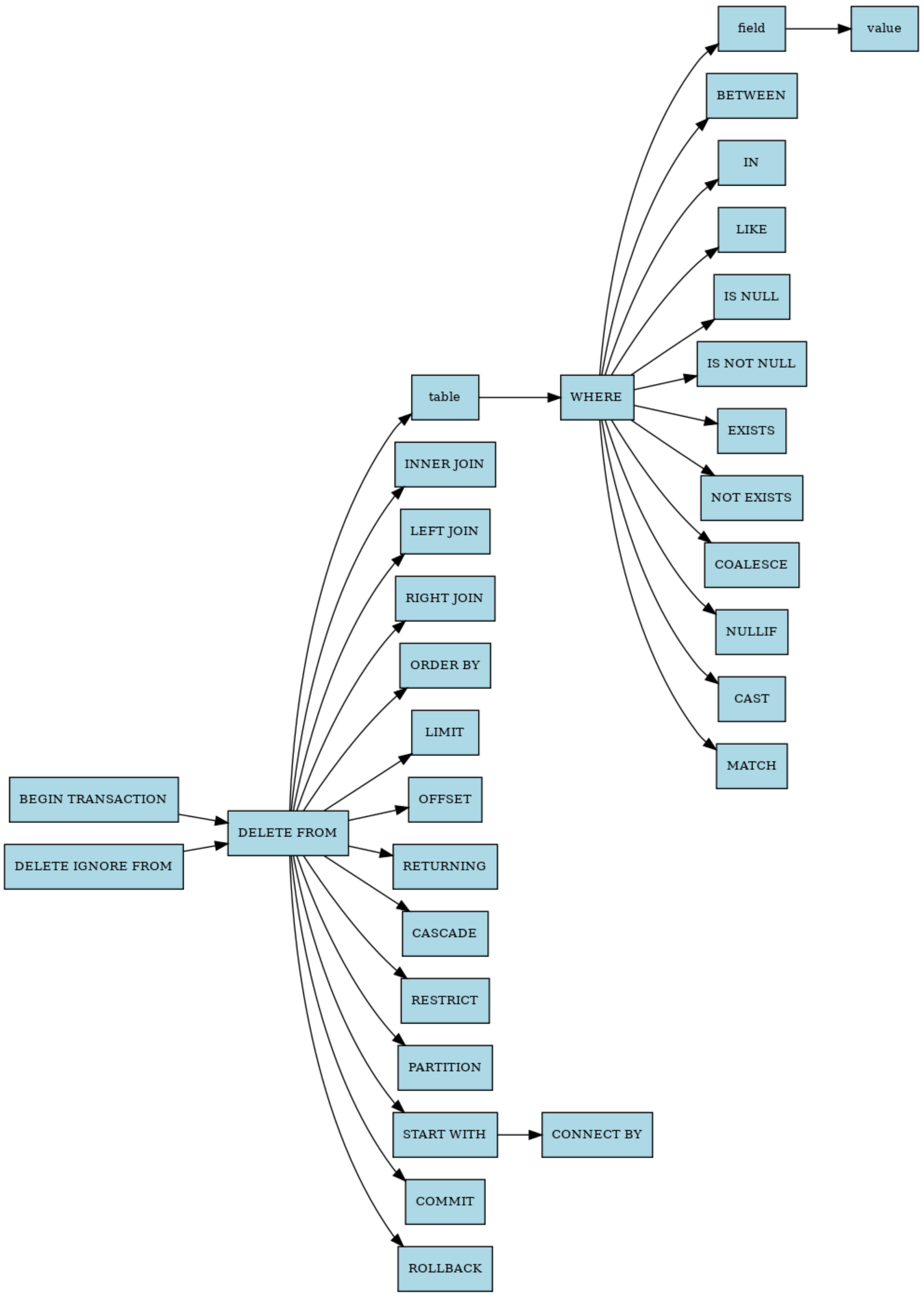


Figure 3.7: Knowledge graph of Delete

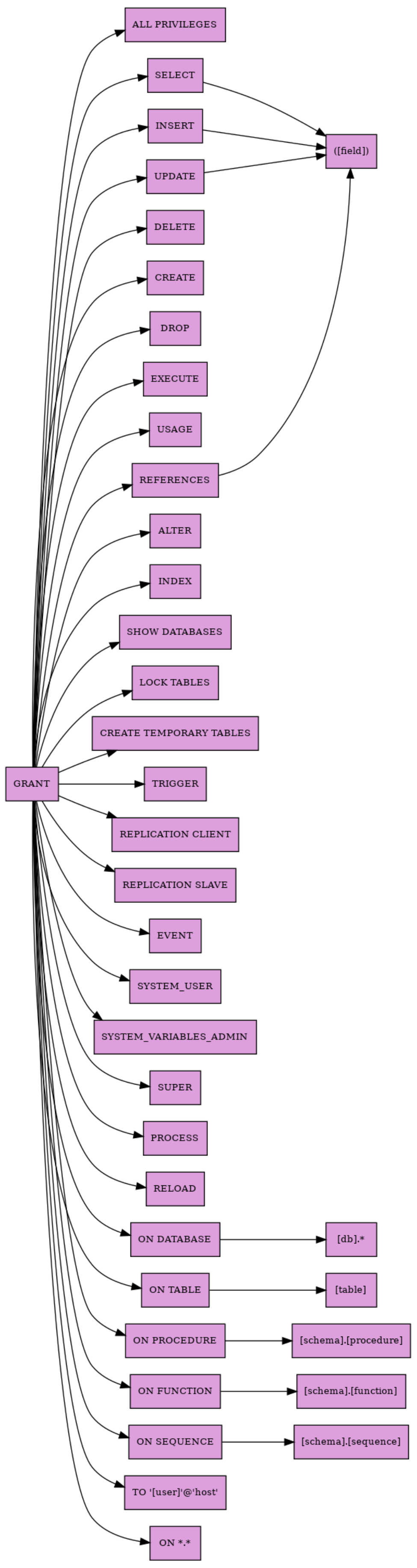


Figure 3.8: Knowledge graph of Grant

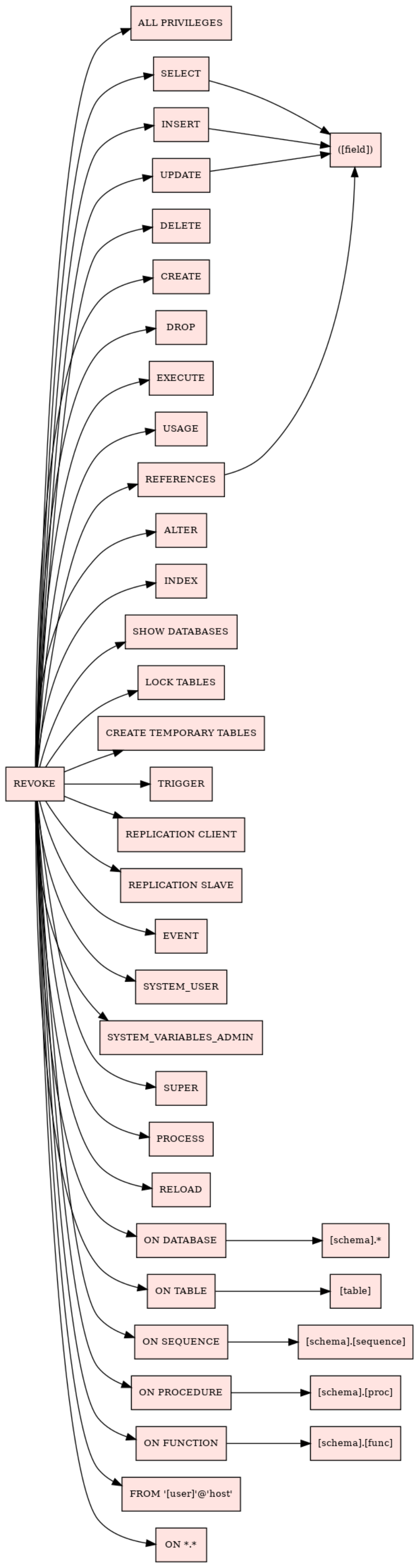


Figure 3.9: Knowledge graph of Revoke

| Type | Command | Description | Example |
|---|---|---|---|
| DDL | CREATE | Creates a new table, database, view, etc. | `CREATE TABLE Students (ID INT, Name VARCHAR(50));` |
|  | ALTER | Modifies an existing database object | `ALTER TABLE Students ADD Age INT;` |
|  | DROP | Deletes an existing database object | `DROP TABLE Students;` |
| DML | SELECT | Retrieves data from the database | `SELECT * FROM Students;` |
|  | INSERT | Inserts new data into a table | `INSERT INTO Students (ID, Name) VALUES (1, 'John');` |
|  | UPDATE | Updates existing data in a table | `UPDATE Students SET Name = 'Johnny' WHERE ID = 1;` |
|  | DELETE | Deletes data from a table | `DELETE FROM Students WHERE ID = 1;` |
| DCL | GRANT | Gives privileges to users | `GRANT SELECT, INSERT ON Students TO user_name;` |
|  | REVOKE | Takes back privileges from users | `REVOKE INSERT ON Students FROM user_name;` |

Table 3.1: Knowledge Graph: SQL Commands Overview

## 3.3 Dataset And Preparation

In this project, the general SQL queries were constructed using a well-structured ontology that visually and logically maps the relationships between different SQL keywords and clauses. Each major SQL command—such as SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, GRANT, and REVOKE—was broken down into its component parts and connected through a keyword-to-keyword flow resembling a "spider trap" diagram. These ontology graphs were initially defined as line-based hierarchies and then converted into visual knowledge graphs with the help of tools like ChatGPT for relationship mapping and Protégé for building formal ontologies. This foundation helped us standardize query structures in our generalize_queries.txt database, allowing queries to be generated dynamically by following the logical path laid out in the ontology—for example, from SELECT to FROM, to WHERE, and then to functions like GROUP BY, ORDER BY, or window functions. These interconnections guided how each SQL clause could be used and combined, ensuring that the queries remain syntactically correct and contextually relevant across use cases. The ontology served not just as a reference model but also as the core blueprint for query generation and interpretation.
QUeries are manually created the generalize/queries file Figure: 3.10 by closely following the ontology diagrams we designed for various SQL commands. These ontology graphs displayed the logical flow of SQL syntax through directional arrows—starting from a root keyword like SELECT, INSERT INTO, UPDATE, or CREATE, and branching out to related clauses and expressions. Each arrow represented a syntactic or semantic relationship between keywords, guiding how one component connects to another—for instance, SELECT pointing to FROM, which then led to WHERE, GROUP BY, and so on. By carefully observing this flow, we con-

structed generalized versions of SQL queries, ensuring each line followed the structure laid out by the ontology. This process helped us maintain consistency, clarity, and correctness while abstracting complex SQL statements into a standardized form. Tools like and supported us in understanding and visualizing these relationships, but the actual query formulation was done manually, step by step, by interpreting the ontology line connections and translating them into real query logic.

A total of approximately 1400 SQL queries were collected and organized during the development of the EASYSQL system. These queries were distributed across various SQL operation types, ensuring wide coverage and diversity. Among them, over 300 queries were related to SELECT (DQL) operations, around 150 queries involved INSERT (DML), approximately 120 queries were for UPDATE (DML), and about 130 queries were for DELETE (DML). Additionally, more than 200 queries covered CREATE (DDL) operations, while DROP (DDL) and ALTER (DDL) queries contributed around 180 and 120 entries respectively. GRANT and REVOKE (DCL) operations accounted for nearly 100 queries. This well-structured dataset ensured that the EASYSQL system could provide real-time, accurate suggestions across all major SQL commands, enhancing its usability and learning support for users. Below are some sample queries in Figure: 3.10.

There are very few requrements and also support various operation systems including Windows 10, Windows 11, Ubuntu (versions 20.04 LTS and 22.04 LTS), and Manjaro Linux 25.0.0. The application is compatible with Python versions 3.6 and above [2], and was successfully tested in environments running Python 3.6, 3.10, and 3.13.2. The project primarily uses the Python library prompt_toolkit (version 3.0.51)[3], along with standard libraries such as os, re, and pytest-shutil 1.8.1 [4], rich 14.0.0 [5], EASYSQL operates through terminal environments like Windows Command Prompt and Linux Terminals like ZSH, BASH. Database connections were tested with both MySQL [6] and 11.7.2-MariaDB [7] servers . Minimum recommended system requirements include at least 2 GB RAM, a functioning Python environment, and access to a database server.

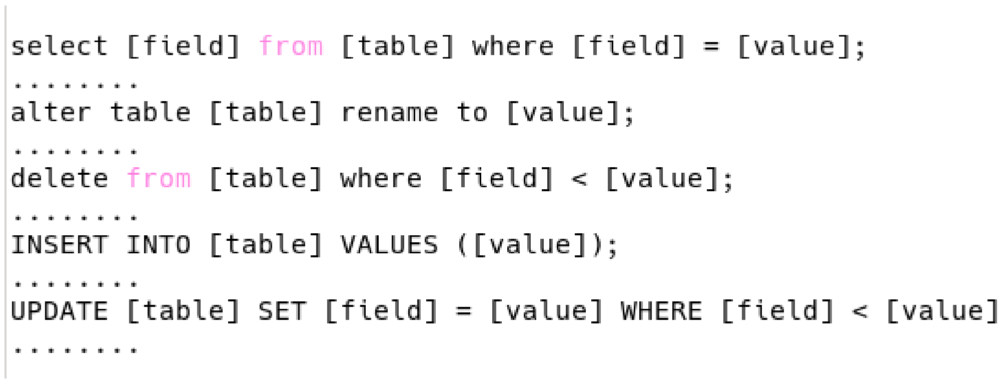

```
select [field] from [table] where [field] = [value];
........
alter table [table] rename to [value];
........
delete from [table] where [field] < [value];
........
INSERT INTO [table] VALUES ([value]);
........
UPDATE [table] SET [field] = [value] WHERE [field] < [value]
........
```

Figure 3.10: Sample Queries

[2]https://www.python.org/downloads/
[3]https://pypi.org/project/prompt-toolkit/
[4]https://docs.python.org/3/library/shutil.html
[5]https://rich.readthedocs.io/en/stable/introduction.html
[6]https://www.mysql.com/
[7]https://mariadb.org/

## 3.4 Detailed Methodology

The Detailed Methodology section explains the full internal working of the EASYSQL system. It covers the overall schema, the main functional modules, and how the flow starts from database connection, user query input, real-time suggestions, validation, correction, and final execution. The system is designed to guide users interactively and help them write correct SQL queries based on the live database schema. In the background, EASYSQL prepares a token list from stored SQL queries. As the user types, the system splits the input into complete and incomplete tokens and predicts the next words using token matching and schema information.

### 3.4.1 Overall Schema

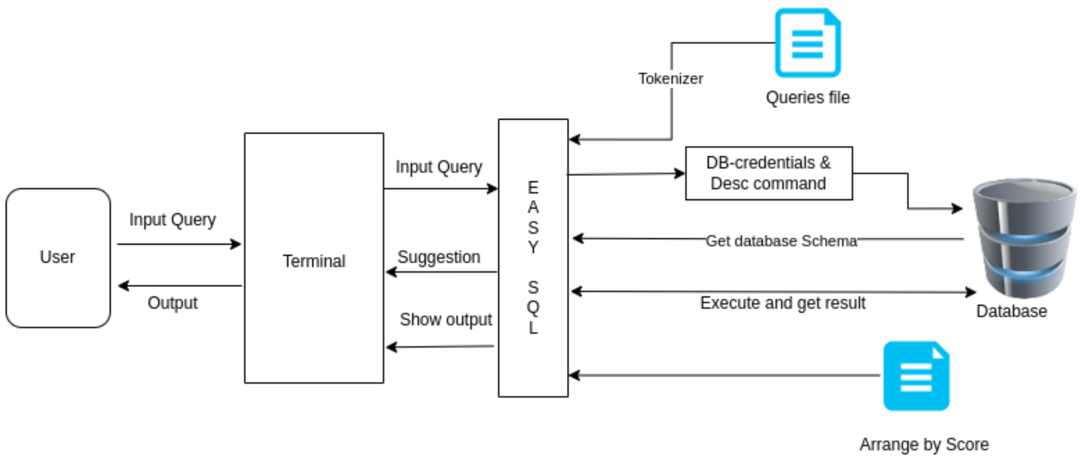


Figure 3.11: Workflow Diagram

1. User Input The user writes and submits an SQL query through the **Terminal interface**.
2. Query Forwarding The Terminal forwards the input query to the **EasySQL engine**, which serves as the core processing unit.
3. **Accessing Pre-existing Queries**
    - EasySQL accesses a **Queries file**.
    - A **Tokenizer module** processes this file to break down and analyze existing query patterns for better suggestions and validations.
4. **Fetching Database Schema**
    - EasySQL uses stored **Database Credentials** and sends a **DESC (describe)** command to the connected **Database**.
    - The system fetches the **database schema**, including table structures and data types.
5. **Query Analysis and Suggestion Generation**
    - With both the input query and the database schema in hand, EasySQL analyzes the input.

- Suggestions are generated based on pre-existing query patterns and the database schema.
- These suggestions are **arranged by scores**, which could be based on relevance, frequency, or contextual accuracy.

6. **Providing Suggestions to User**

   - The **scored suggestions** are sent back to the **Terminal**.
   - The user can view these suggestions and optionally modify their original query.

7. **Query Execution**

   Once the final query is confirmed (original or modified), EasySQL **executes the query** against the **Database**.

8. **Fetching and Displaying Results**

   - The execution result is retrieved from the database.
   - EasySQL sends the output to the Terminal.
   - The Terminal displays the **final output** to the **user**, completing the cycle.

9. **System Role Summary**

   Throughout the workflow, EasySQL acts as a smart intermediary:

   - Providing **real-time suggestions**.
   - **Validating** queries based on schema.
   - **Enhancing usability** and **reducing errors** for users writing SQL queries.

### 3.4.2 Backend steps:

This diagram 3.12 outlines how user queries are tokenized, predicted, and executed inside EASYSQL.

1. **User Input and Tokenization**
   The process begins when a user enters a natural-language or semi-structured query into the system interface. This input is then passed through a tokenizer, which splits the sentence into individual tokens—essentially breaking the query down into meaningful words or components. These tokens serve as the foundation for the system to understand the user's intent.

2. **Token Prediction and Memory Learning**
   Next, the tokenized input is analysed by a prediction module that identifies which of the tokens form complete, meaningful elements. This module uses a memory component that stores previously learned SQL structures and patterns. The memory is continually updated by a tokenizer trained on a structured Queries File, which contains various standard SQL examples. This allows the system to improve its predictions over time.

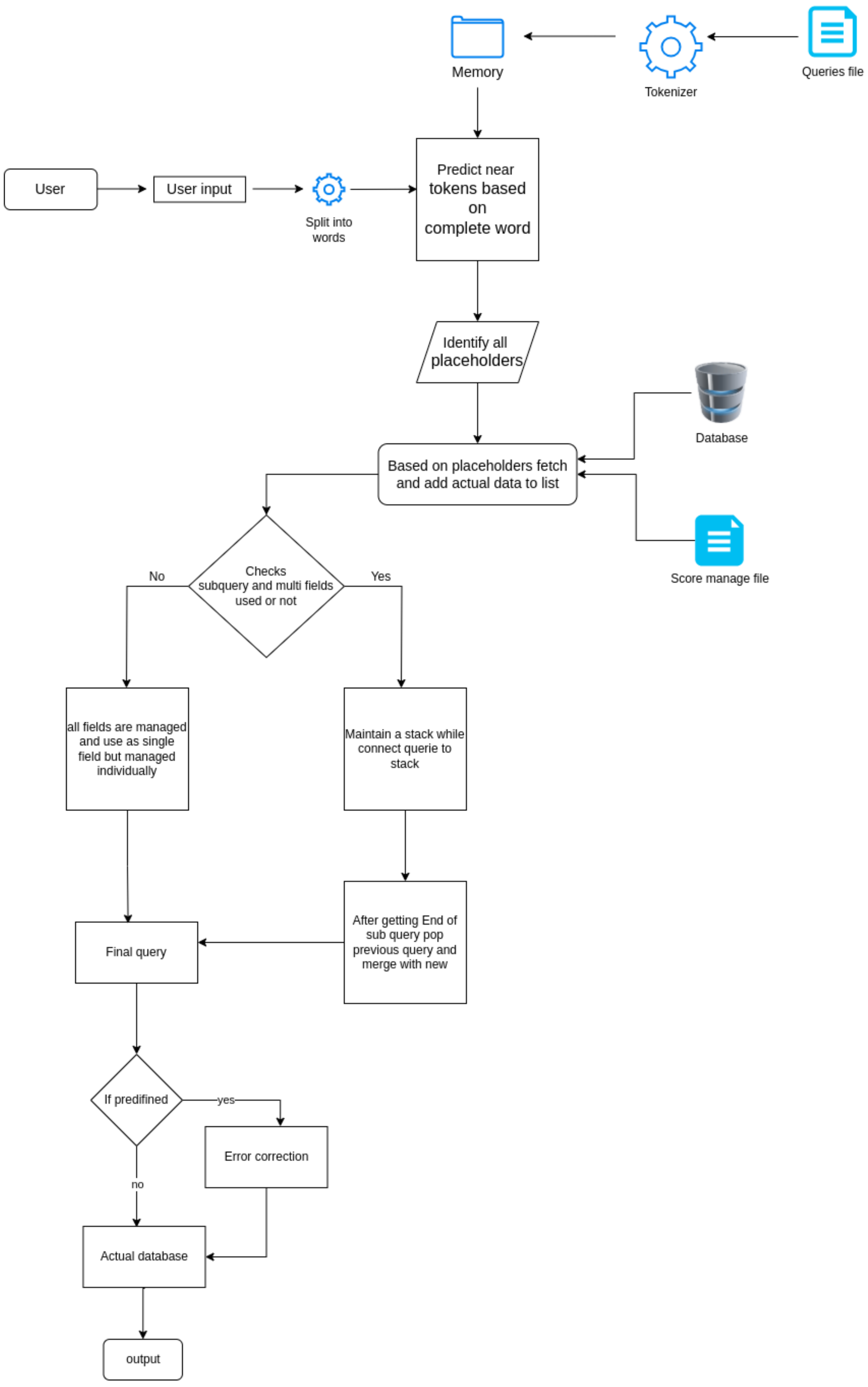

Memory
Tokenizer
Queries file
User
User input
Split into words
Predict near tokens based on complete word
Identify all placeholders
Database
Based on placeholders fetch and add actual data to list
Score manage file
No
Checks subquery and multi fields used or not
Yes
all fields are managed and use as single field but managed individually
Maintain a stack while connect querie to stack
After getting End of sub query pop previous query and merge with new
Final query
If predifined
yes
Error correction
no
Actual database
output

Figure 3.12: Overall steps of the EASYSQL system

3. **Placeholder Detection and Context Awareness**
   Once complete tokens are recognized, the system scans for placeholders like [table], [field], or [condition] that represent parts of the query needing dynamic, schema-specific content. These placeholders are identified for replacement using actual database values and context.

4. **Schema Access and Score-Based Replacement**
   The system then interacts with the actual database and a score management file to fetch relevant table names, field names, and conditions. These values are not chosen randomly; instead, they are ranked and selected based on frequency, contextual relevance, and similarity scoring. This ensures that the placeholder replacements are both accurate and meaningful.

5. **Query Complexity Handling**
   At this stage, the system checks if the query involves simple field references or more complex structures like subqueries. If the query is straightforward, all fields are treated as independent units and merged into a final SQL statement. If subqueries or multiple field groups are present, a stack-based mechanism is used. Subqueries are pushed onto a stack as they are processed and later popped and merged with the main query once complete, ensuring logical and syntactical consistency.

6. **Final Query Generation**
   Once all components—tokens, placeholders, and subqueries—are resolved and structured, the system constructs the final SQL query. This query represents the closest possible version of what the user intended to write, formatted correctly and complete.

7. **Predefined Query Check and Error Correction**
   Before execution, the system checks whether the generated query matches a predefined query from its library. If it does, the query is sent through an error correction module to fix any minor syntax issues or mismatches. If it doesn't match a predefined template, the query skips this step and proceeds directly to execution.

8. **Query Execution and Output**
   The final query is executed on the connected database. Once processed, the result is fetched and displayed back to the user through the interface, completing the interactive query-building loop.

**Backend of Tokenization**

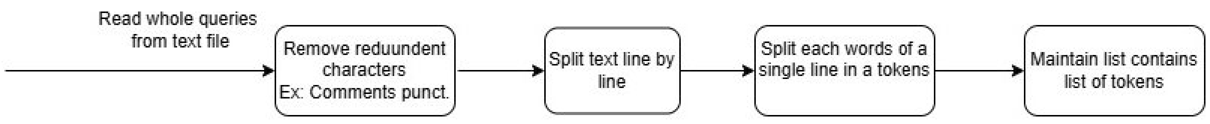

Read whole queries from text file
Remove reduundent characters Ex: Comments punct.
Split text line by line
Split each words of a single line in a tokens
Maintain list contains list of tokens

Figure 3.13: Tokenization

**Tokenization Phase – Query Preprocessing & Token Management**

1. **Read Queries**: The system begins by reading a list of predefined SQL queries from a plain text file.

2. **Clean Queries**: It removes redundant elements such as comments and extra punctuation to reduce noise.
3. **Split by Line**: The cleaned queries are split line by line to isolate each SQL statement.
4. **Tokenize Lines**: Each line is further broken down into individual words (tokens).
5. **Maintain Token List**: The tokens from each line are stored in a structured list format (token dictionary).

**Send to Memory**: The tokenized structures are passed to the **Memory Module** for future reference during suggestions.

**Suggestion Phase – Real-time Query Assistance**

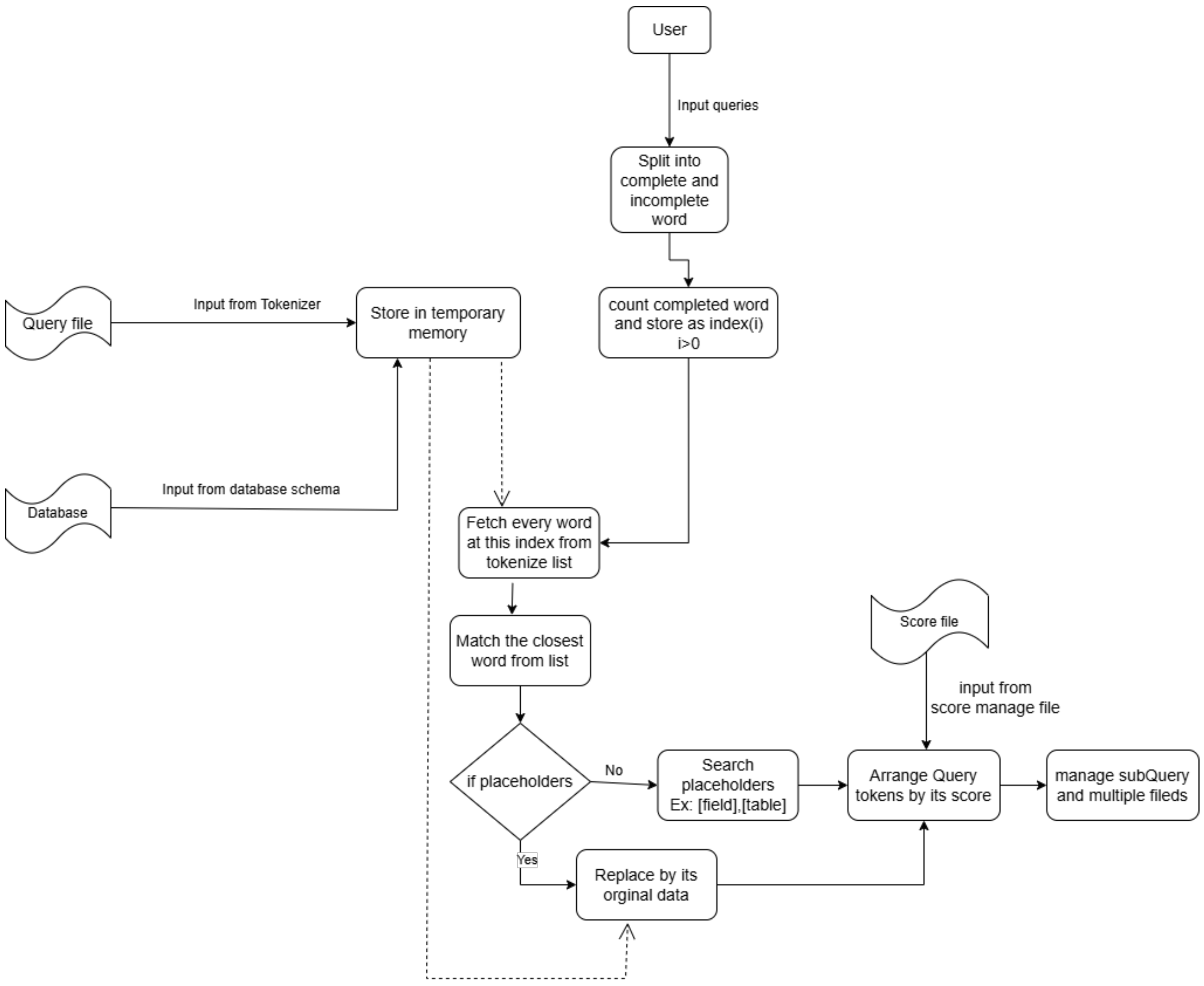

User
Input queries
Split into complete and incomplete word
count completed word and store as index(i) i>0
Query file
Input from Tokenizer
Store in temporary memory
Database
Input from database schema
Fetch every word at this index from tokenize list
Match the closest word from list
if placeholders
No
Yes
Search placeholders Ex: [field],[table]
Score file
input from score manage file
Arrange Query tokens by its score
manage subQuery and multiple fileds
Replace by its orginal data

Figure 3.14: Suggestion Process

1. **User Input**: A user provides a partial or incomplete query (e.g., "SELECT name FR").
2. **Token Analysis**: The system splits the input into complete and incomplete tokens. (e.g., SELECT and name are complete; FR is incomplete).
3. **Count Complete Tokens**: It maintains a count (i) of complete tokens to fetch the next expected token.

4. **Temporary Storage**: The system stores tokens from the **Tokenizer** and schema info in temporary memory.
5. **Next Token Prediction**: It fetches the next token from the token list at index i and matches it with the incomplete input to predict the most likely word (e.g., FROM).
6. **Placeholder Handling**: If placeholders like [field] or [table] are found in the predicted token:
   - It fetches the actual table or column names from the **Database Schema**.
   - Replaces placeholders with actual field names (e.g., [field] $\rightarrow$ name).

**Score Management & Query Structuring**

1. **Score-Based Ranking**: The **Score Management File** ranks suggestions based on frequency, similarity, or context.
2. **Suggestion Filtering**: Only the most relevant, high-scoring suggestions are presented to the user.
3. **Subquery & multi-Field Logic**:
   - If subqueries or multiple fields are involved, they are processed modularly.
   - Subqueries are pushed onto a stack, and popped when completed to merge logically.

**Finalization & Execution**

1. **Final Query Construction**: All predicted and matched tokens are combined into a complete final SQL query.
2. **Predefined Check**: The system checks if the query matches a known predefined query:
   - **If Yes**: It undergoes **error correction** to fix any anomalies.
   - **If No**: It is sent **directly to the database** for execution.
3. **Display Output**: The result from the database is fetched and displayed to the user.

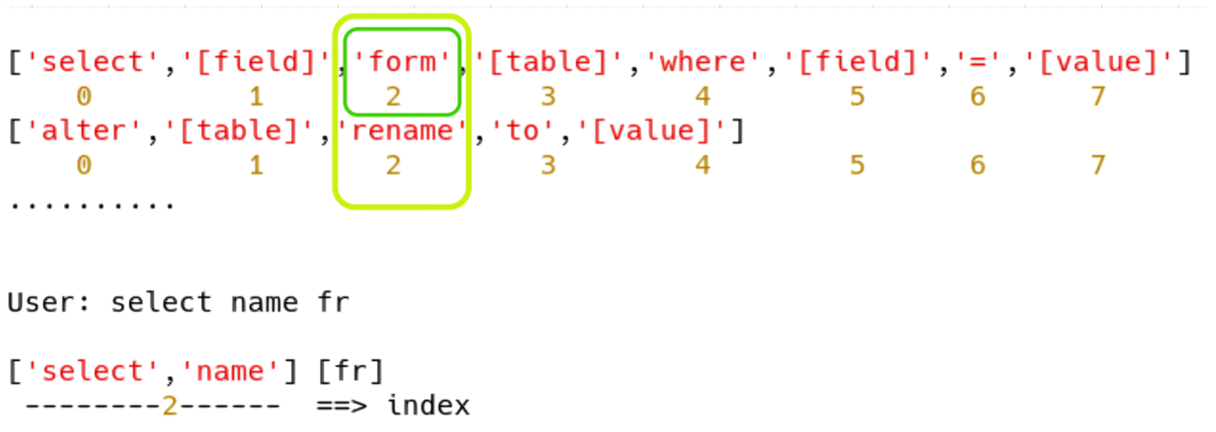
# Chapter 4

# Result and Discussion

In this section, Evaluation of the system's ability to provide real-time SQL suggestions, detect and correct errors, and assist users in learning SQL syntax and structure is planned for future stages. Key aspects such as prediction accuracy, context-awareness, handling of complex queries, and overall user experience will be considered during the evaluation. Testing will involve a variety of sample queries and schema configurations to reflect common scenarios faced by beginner-level SQL learners.

## 4.1 Query suggestion steps:

At first, fetch the text queries from the text file and remove the comments, punctuations, and redundant characters, then make it similar case and then split into lines. Further, each line will be broken into small tokens, thus tokenization will be done and stored in memory. Similarly, get the database schema using database credentials like table names, their fields, their types, etc., and also store into memory. Now, when the user starts inputting the text at runtime, break it into two parts: one is a list which will collect completed words (tokens), and another is the incomplete word. Now count the total completed words; it will be the index (i > 0) of the next word from the tokenized list. Now match previous completed words to tokenized list's each line—if any line is matched, then the index (i) will be stored in the suggestion list. Then match with the incomplete word. If it is not null (not empty), then it means the user is trying to type something. Now match this incomplete word within the suggestion list words (tokens) and create the final suggestion list like figure 4.1. Now, if it is a placeholder, then it again finds corresponding database actual data for this place and also updates the suggestion list. Now finally, arrange those suggestion tokens based on score, which is stored in a score manager file. In between that, for subquery we used a stack and for fields we used an internal merging mechanism.



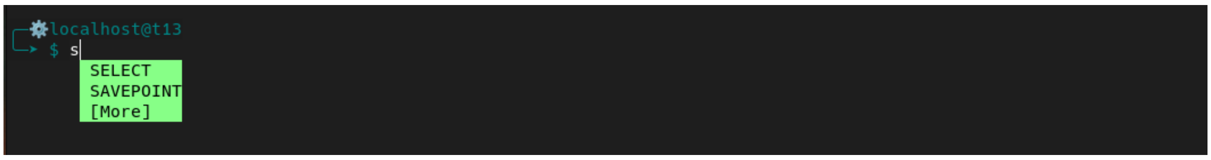
```
['select','[field]','form','[table]','where','[field]','=','[value]']
    0         1        2       3        4        5      6      7
['alter','[table]','rename','to','[value]']
    0         1        2       3        4        5      6      7
..........


User: select name fr

['select','name'] [fr]
 --------2------   ==> index
```

Figure 4.1: Find Process

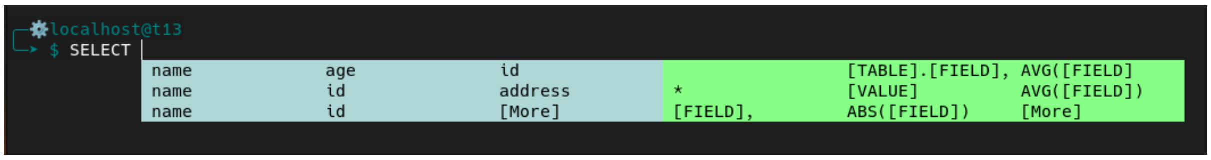
In our system, the user first inputs a normal SQL query by typing directly into the terminal. As the user types, the sentence is continuously divided into two portions: one part containing completed tokens (which are stored in a list) and another part containing the currently incomplete word being typed. For example, if the user intends to write a DQL (Data Query Language) query like SELECT, and starts typing with the letter s, the system immediately identifies s as an incomplete token and searches through the tokenized list for suggestions that start with s, displaying possible completions for the user to select. (4.2)



```
┌─localhost@t13
└─› $ s
      SELECT
      SAVEPOINT
      [More]
```

Figure 4.2: Step 1

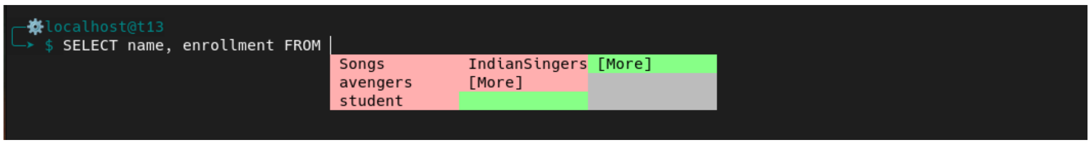
Once the user completes the word (for example, SELECT) and presses the space key, the system then moves to predict the next possible words. It intelligently matches the next token, which could be a field name or a function, and provides appropriate suggestions to assist the user in writing the query faster. (4.3)



```
┌─localhost@t13
└─› $ SELECT
       name        age         id                                  [TABLE].[FIELD], AVG([FIELD]
       name        id          address       *                     [VALUE]          AVG([FIELD])
       name        id          [More]        [FIELD],              ABS([FIELD])     [More]
```

Figure 4.3: Step 2

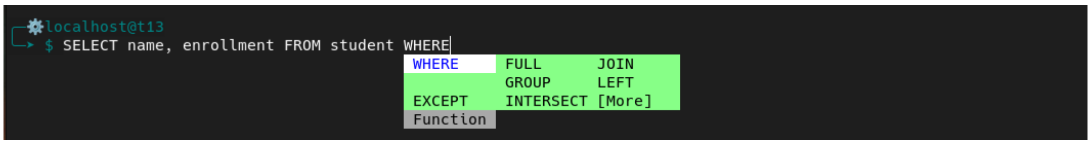
If the query involves placeholders like [fields],[table] for column name or table's name, the system automatically fetches real column names from the connected database, such as name, enrollment, etc., and replaces the placeholders with actual field names. Similarly, after typing keywords like FROM, the tool queries the database schema to fetch and suggest actual table names to the user. (4.4)



```
┌─localhost@t13
└─› $ SELECT name, enrollment FROM
                                   Songs        IndianSingers [More]
                                   avengers     [More]
                                   student
```

Figure 4.4: Step 3

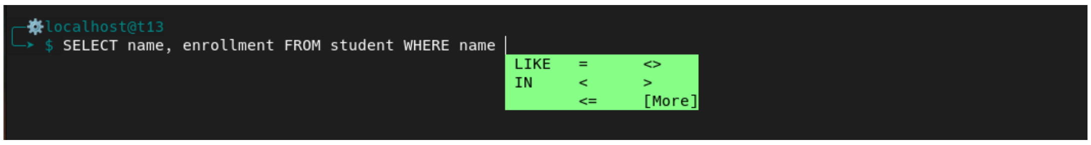
As the user continues building the query, if they wish to extend it further with additional clauses, the system dynamically offers suggestions for SQL clauses such as WHERE, HAVING, IN, and others based on the tokenized list. (4.5)



```
┌─localhost@t13
└─› $ SELECT name, enrollment FROM student WHERE
                                          WHERE     FULL      JOIN
                                                    GROUP     LEFT
                                          EXCEPT    INTERSECT [More]
                                          Function
```

Figure 4.5: Step 4

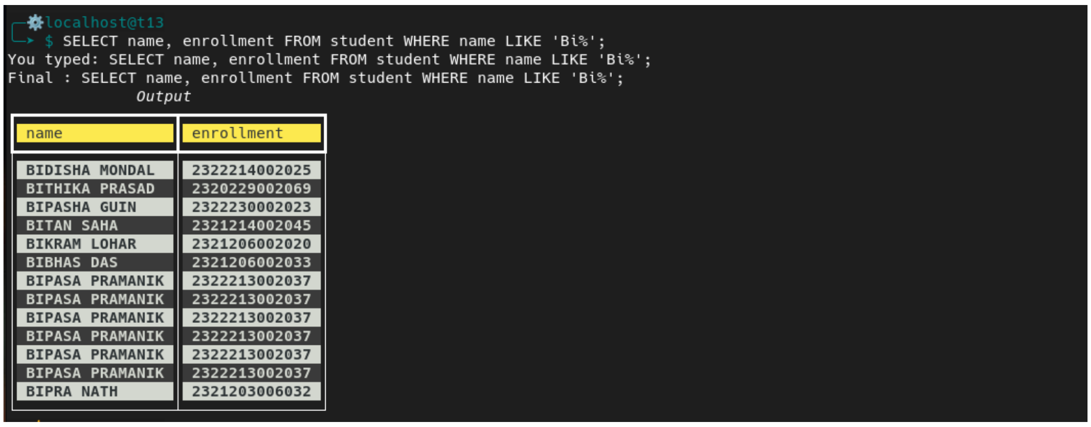
After clauses like WHERE, the tool assists the user by suggesting field names again along with appropriate condition operators. If the user forgets to end the query with a semicolon ;, the system automatically corrects this and ensures the query structure remains valid.(4.6)



```
localhost@t13
$ SELECT name, enrollment FROM student WHERE name
                                                 LIKE   =      <>
                                                 IN     <      >
                                                        <=     [More]
```

Figure 4.6: Step 5

If user want to terminate this query,it is passed through multiple validation stages and, upon successful validation like ';' adding keyword cheking, is executed on the database. The result of the query execution is then displayed to the user directly on the terminal, providing a seamless and intelligent experience (4.7).



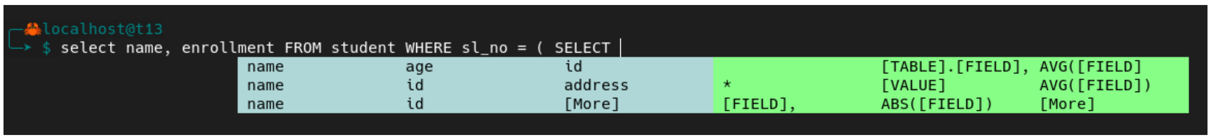
```
localhost@t13
$ SELECT name, enrollment FROM student WHERE name LIKE 'Bi%';
You typed: SELECT name, enrollment FROM student WHERE name LIKE 'Bi%';
Final : SELECT name, enrollment FROM student WHERE name LIKE 'Bi%';
                 Output

 name               enrollment

 BIDISHA MONDAL     2322214002025
 BITHIKA PRASAD     2320229002069
 BIPASHA GUIN       2322230002023
 BITAN SAHA         2321214002045
 BIKRAM LOHAR       2321206002020
 BIBHAS DAS         2321206002033
 BIPASA PRAMANIK    2322213002037
 BIPASA PRAMANIK    2322213002037
 BIPASA PRAMANIK    2322213002037
 BIPASA PRAMANIK    2322213002037
 BIPASA PRAMANIK    2322213002037
 BIPASA PRAMANIK    2322213002037
 BIPRA NATH         2321203006032
```

Figure 4.7: Step 6

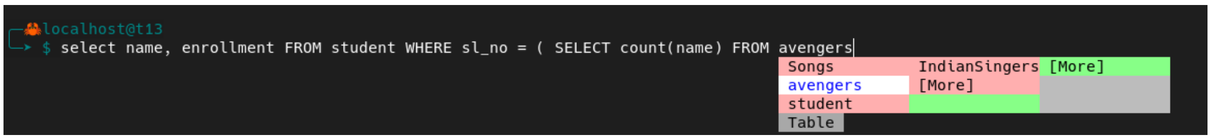
If the user wishes to utilize a subquery, it is easy for them to do so. For instance, once the user has typed a WHERE clause, the user can initiate a subquery by typing SELECT, as indicated in (Figure [IMAGE]). At this point, the system handles the subquery as a new query: the prior main query is pushed onto the stack, and all tokenization, suggestions, and checks proceed normally for the subquery. (4.8)



```
localhost@t13
$ select name, enrollment FROM student WHERE sl_no = ( SELECT
                         name        age         id                               [TABLE].[FIELD], AVG([FIELD]
                         name        id          address       *                  [VALUE]          AVG([FIELD])
                         name        id          [More]        [FIELD],           ABS([FIELD])     [More]
```

Figure 4.8: Step 7

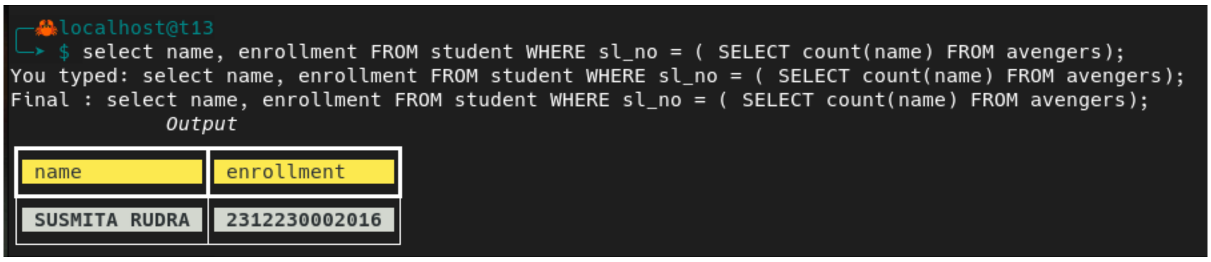
When a closing bracket ) is encountered, the system knows that the subquery has terminated. It then pops the last main query from the stack, appends the subquery properly to it, and proceeds to handle it as a single complete query, with proper SQL structure and flow.(4.9)



```
localhost@t13
$ select name, enrollment FROM student WHERE sl_no = ( SELECT count(name) FROM avengers
                                                                                Songs      IndianSingers [More]
                                                                                avengers   [More]
                                                                                student
                                                                                Table
```

Figure 4.9: Step 8

Once the entire query is created a simple query or a subquery involving one or more the system itself corrects minor errors such as missing semicolons (;) at the end of a statement. It checks the complete SQL query through various internal stages to confirm that there are no syntax errors figure: 4.10.



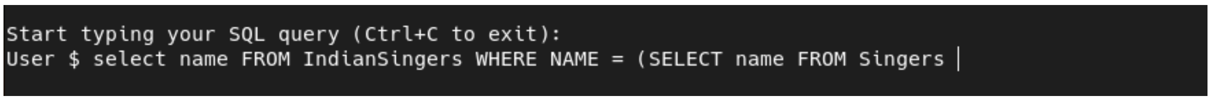
```
┌─localhost@t13
└─➤ $ select name, enrollment FROM student WHERE sl_no = ( SELECT count(name) FROM avengers);
You typed: select name, enrollment FROM student WHERE sl_no = ( SELECT count(name) FROM avengers);
Final : select name, enrollment FROM student WHERE sl_no = ( SELECT count(name) FROM avengers);
             Output
 name            enrollment
 SUSMITA RUDRA   2312230002016
```

Figure 4.10: Step 9

### 4.1.1 Challenges Faced During Development

#### Subquery's Problem

The initial query dataset file contained queries with arbitrary numbers of subqueries. Due to this, our program could only suggest subquery patterns up to the extent of what was available in the file. When users tried to insert deeper or multiple subqueries that were beyond the predefined extent, the system did not find matching ones from the tokenized list, and it appeared to be hardcoding. This hindered the processing of nested queries correctly figure: 4.11.

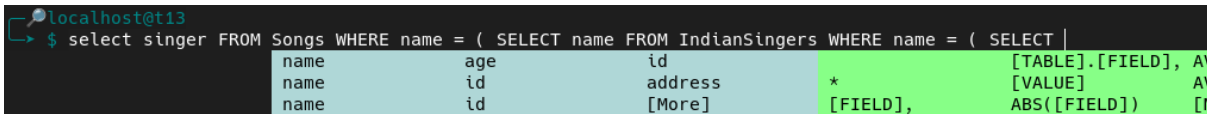


```
Start typing your SQL query (Ctrl+C to exit):
User $ select name FROM IndianSingers WHERE NAME = (SELECT name FROM Singers |
```

Figure 4.11: Subquery problem

#### Proposed solution:

Subqueries are always enclosed in parentheses and begin with the `SELECT` keyword. Based on this structure, a stack-based logic is implemented. Whenever a subquery is detected, it is pushed onto a stack and treated as the current query. When a closing parenthesis ) is encountered, the subquery is considered complete. It is then popped from the stack, treated as a single `[value]`, and merged back into the main query. This approach allows the simple query model to handle nested subquery structures effectively figure: 4.12.

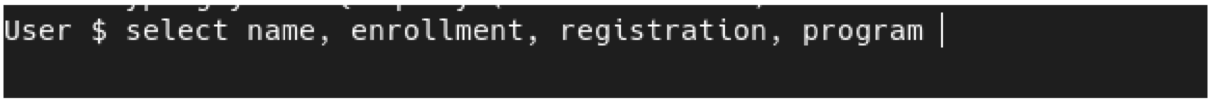


```
┌─localhost@t13
└─➤ $ select singer FROM Songs WHERE name = ( SELECT name FROM IndianSingers WHERE name = ( SELECT |
                         name         age          id                                 [TABLE].[FIELD], A
                         name         id           address          *                 [VALUE]          A
                         name         id           [More]           [FIELD],          ABS([FIELD])     [
```

Figure 4.12: Subquery solution

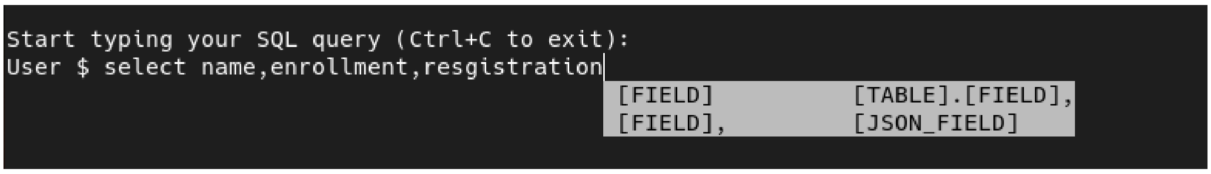
#### Space-Related Issues

Previous query dataset had queries with a randomly chosen number of fields. This permitted our program to make suggestions accurately up to the maximum number of fields that are available. But when a user attempted to include additional fields beyond what was available in the query file, the program was no longer able to match from the tokenized list, causing it to act like hard-coded suggestions figure : 4.13 . Besides that, if one user entered more than one field without spaces added after commas (SELECT name,id,address), the system treated the whole section (name,id,address) as a single field rather than individual fields figure : 4.14 Due to this, whenever the user attempted to hit Tab or other keys for suggestions, it would delete or misprocess the whole group, as it processed it as a single token.



```
User $ select name, enrollment, registration, program
```

Figure 4.13: Space problem 1

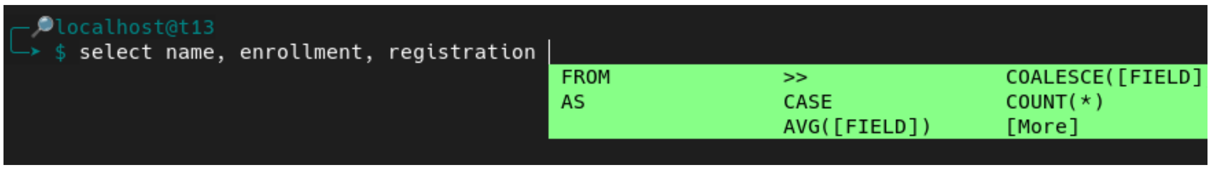


```
Start typing your SQL query (Ctrl+C to exit):
User $ select name,enrollment,resgistration
                                            [FIELD]          [TABLE].[FIELD],
                                            [FIELD],         [JSON_FIELD]
```

Figure 4.14: Space problem 2

**Proposed solution**

All random multiple-fields queries were filtered out from the query set, retaining only single-field queries. During user input, if multiple fields are typed, a space (' ') is dynamically added after every comma (,) so that the fields are regarded as separate words. This allows the program to apply suggestions correctly for each field separately. Internally, once we finish suggesting each for one field separately, we combine them as a single field, so even with single-field queries only in the dataset, the system is still able to make accurate suggestions for multiple fields in a query figure : 4.15.



```
┌─localhost@t13
└─> $ select name, enrollment, registration
                                            FROM          >>              COALESCE([FIELD]
                                            AS            CASE            COUNT(*)
                                                          AVG([FIELD])    [More]
```

Figure 4.15: Space Solution

## 4.2 Query Correction

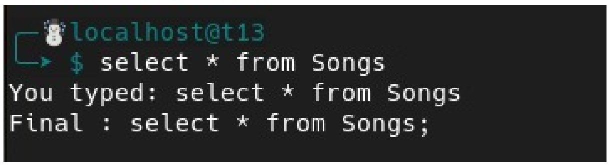
While writing SQL queries, users often make simple but important mistakes, such as forgetting to add a semicolon (;) at the end of a statement, missing closing parentheses when using grouped conditions, or typing incorrect SQL keywords due to spelling mistakes. These errors can cause the query to fail during execution.

To solve these problems, EASYSQL provides an intelligent real-time query correction system. The system automatically:

- Appends a missing semicolon (;) at the end of a query if the user forgets to type it.
- Detects unmatched opening parentheses '(' and adds the missing closing parentheses ')' automatically.
- Corrects common keyword spelling mistakes (for example, "SELEC" corrected to "SELECT") using fuzzy matching.

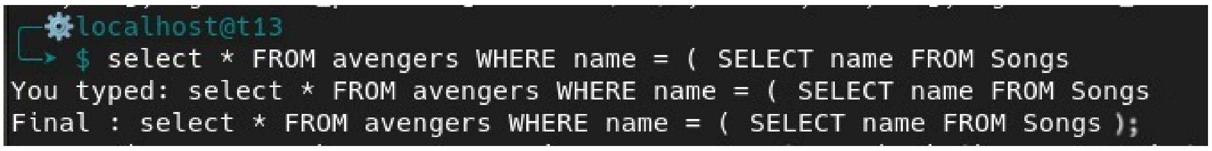
These corrections are applied during typing without interrupting the user, making the system more user-friendly and improving the learning process.

Figures 4.16 and 4.17 show examples of automatic correction of missing semicolon and missing parenthesis. (Spelling correction is handled internally but not shown separately.)



```
┌─localhost@t13
└─> $ select * from Songs
You typed: select * from Songs
Final : select * from Songs;
```

Figure 4.16: Correction 1



```
┌─localhost@t13
└─> $ select * FROM avengers WHERE name = ( SELECT name FROM Songs
You typed: select * FROM avengers WHERE name = ( SELECT name FROM Songs
Final : select * FROM avengers WHERE name = ( SELECT name FROM Songs );
```

Figure 4.17: Correction 2

# Chapter 5

# Conclusion

The query suggestion system, which began as a simple tokenizer and placeholder-based query generator, has now evolved into a functional application designed to assist users—especially beginners—in constructing valid SQL queries in real time. By processing pre-defined query structures, tokenizing them, and dynamically replacing placeholders like [field] and [table] with actual schema values, this tool significantly reduces the chances of syntactic errors and accelerates query building.

Through intuitive design and efficient string matching, the system provides real-time suggestions as users type. It helps in maintaining context, suggesting not just fields and tables but also full queries based on partial input. What started as a static idea is now an interactive, user-friendly application that bridges the gap between database structure and query formulation—making SQL easier for all users.

# Chapter 6

# Future Scope

The future scope of EasySQL in terms of Machine Learning (ML) integration envisions transforming it into a more intelligent, adaptive SQL assistant. Lightweight models such as Decision Trees, Random Forests, and Logistic Regression can be incorporated to improve token prediction accuracy and enhance the ranking of query suggestions based on user behavior and schema context. Clustering algorithms like K-Means can be employed to detect frequently used query patterns, allowing EasySQL to suggest commonly executed queries or templates dynamically. Reinforcement Learning techniques, such as Q-learning, can further enable the system to adapt over time by learning from user feedback, refining its suggestions with each interaction. Additionally, building contextual ranking models will allow EasySQL to prioritize suggestions by analyzing table relationships, field types, and historical queries. Personalized user profiles can be created by training models to understand individual user habits and commonly accessed tables, offering a customized experience. Finally, implementing anomaly detection methods, such as Isolation Forests, can help identify and correct unusual or erroneous query patterns, making EasySQL a smarter, self-improving tool over time.

# Bibliography

[1] Rajiv Chopra. *Database Management System (DBMS) A Practical Approach*. S. Chand Publishing, 2010.

[2] Nodira Khoussainova et al. "SnipSuggest: Context-aware autocompletion for SQL". In: *Proceedings of the VLDB Endowment* 4.1 (2010), pp. 22–33.

[3] Javad Akbarnejad et al. "SQL QueRIE recommendations". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1597–1600.

[4] Ju Fan, Guoliang Li, and Lizhu Zhou. "Interactive SQL query suggestion: Making databases user-friendly". In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 351–362.

[5] Izzeddin Gür et al. "Dialsql: Dialogue based structured query generation". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 1339–1349.

[6] Quan Do et al. "Automatic generation of SQL queries". In: *2014 ASEE Annual Conference & Exposition*. 2014, pp. 24–221.

[7] Venkata Vamsikrishna Meduri, Kanchan Chowdhury, and Mohamed Sarwat. "Evaluation of machine learning algorithms in predicting the next SQL query from the future". In: *ACM Transactions on Database Systems (TODS)* 46.1 (2021), pp. 1–46.

[8] Anish Doshi, Jianglai Dai, and Min-Chi Chiang. "Synthesizing SQL from Natural Language". In: (2020).

[9] B Balaji Naik et al. "An SQL query generator for cross-domain human language based questions based on NLP model". In: *Multimedia Tools and Applications* 83.4 (2024), pp. 11861–11884.

[10] Hrithik Sanyal, Sagar Shukla, and Rajneesh Agrawal. "Natural language processing technique for generation of SQL queries dynamically". In: *2021 6th International Conference for Convergence in Technology (I2CT)*. IEEE. 2021, pp. 1–6.

[11] Karam Ahkouk et al. "SQLSketch: Generating SQL Queries using a sketch-based approach". In: *Journal of Intelligent & Fuzzy Systems* 40.6 (2021), pp. 12253–12263.

[12] Muhammad Shahzaib Baig et al. "Natural language to sql queries: A review". In: *International Journal of Innovations in Science Technology* 4 (2022), pp. 147–162.

[13] Vincent Toulouse. "Slide-recommendation for improving students SQL solutions using natural language processing". PhD thesis. Otto-von-Guericke University Magdeburg, 2021.

[14] Aditya Sawant et al. "AI Model to Generate SQL Queries from Natural Language Instructions through Voice". In: *Journal of Physics: Conference Series*. Vol. 2273. 1. IOP Publishing. 2022, p. 012014.

[15] Abhishek Kharade. "Generation of sql query using natural language processing". In: *Available at SSRN 4515801* (2023).

[16] Arnav Jha, Naman Anand, and H Karthikeyan. "Conversion of natural language text to SQL queries using generative AI". In: *Hybrid and Advanced Technologies*. CRC Press, 2025, pp. 25–32.