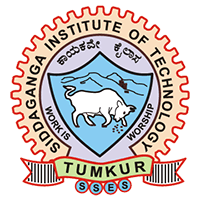
**SIDDAGANGA INSTITUTE OF TECHNOLOGY, Tumakuru- 3**

(An Autonomous institution affiliated to Visvesvaraya Technological University- Belagavi, Approved by AICTE,

Accredited by NAAC with ‘A++’ Grade, Awarded Diamond College Rating by QS I-GAUGE & ISO 9001:2015 certified )



**Natural Language Processing (S6CII01)**

**ABL-2 Microproject Report**

**ON**

**“NL2SQL”**

submitted in the partial fulfilment of the requirements for VII semester,

Bachelor of Engineering in Computer Science and Engineering

By

|  |  |
| --- | --- |
| **Sushant S Mutta** | **1SI22CI054** |
| **Samay Madhusudhan** | **1SI22CI045** |
| **Sharik Sharief** | **1SI22CI047** |

Under the guidance of

**Dr.Srinivasa K**

Assistant Professor

**Department of Computer Science and Engineering**

( Program Accredited by NBA)

**Academic Year: 2024-25 (Even)**

**Table of Contents**

|  |  |
| --- | --- |
| **Contents** | **Page no** |
| **Software and Hardware Environment used for implementation** | **1** |
| **Exposure to the new tool** | **3** |
| **Pseudocode for the modules implemented** | **4** |
| **Experimental Results** | **7** |

**Software Environment**

Our Natural Language to SQL (NL2SQL) project leverages a robust software stack designed for efficient natural language processing, deep learning model development, and seamless database interaction.

**Programming Language**

* **Python:** Chosen as the primary programming language due to its extensive ecosystem of libraries and frameworks crucial for machine learning and natural language processing.

**Natural Language Processing (NLP) Libraries**

* **Hugging Face Transformers:** This library is fundamental to our approach, enabling the utilization of pre-trained Large Language Models (LLMs) such as BERT, GPT, or T5. These models are critical for comprehending the nuances of natural language queries and generating accurate SQL.
* **spaCy:** Employed for efficient, production-ready NLP tasks, including tokenization, part-of-speech tagging, named entity recognition (NER), and dependency parsing. These functionalities are vital for extracting meaningful entities and relationships from user queries.
* **NLTK (Natural Language Toolkit):** Used for various general-purpose NLP tasks, particularly during the research and prototyping phases.

**Machine Learning Frameworks**

* **PyTorch / TensorFlow:** We utilize one of these leading deep learning frameworks (e.g., PyTorch) to build, train, and fine-tune the neural networks that power our NL2SQL conversion engine. This includes fine-tuning the selected LLMs for our specific database schemas.

**Database Technologies**

* **SQL Database (e.g., PostgreSQL / MySQL / SQLite):** The target database where the generated SQL queries will be executed. The specific choice (e.g., PostgreSQL for its robustness and advanced features) depends on the project's data storage and querying requirements.
* **Database Connectors/Drivers:** Python libraries (e.g., psycopg2 for PostgreSQL, mysql-connector-python for MySQL, sqlite3 for SQLite) are used to establish connections between our Python application and the database.
* **SQLAlchemy (Optional):** An SQL toolkit and Object Relational Mapper (ORM) can be optionally used to provide a higher-level, more Pythonic interface for database interactions, enhancing code readability and maintainability.

**Development Tools & Environment**

* **Integrated Development Environment (IDE):** Visual Studio Code (VS Code) or PyCharm for code development, debugging, and project management.
* **Version Control:** Git, integrated with platforms like GitHub, for collaborative development, tracking changes, and managing code versions.
* **Package Manager:** pip (or Conda) for managing Python dependencies and ensuring consistent environments.
* **Containerization (for deployment):** Docker is utilized to package the entire application and its dependencies into isolated containers, ensuring consistent deployment across different environments.

**Supporting Libraries**

* **Pandas:** For efficient data manipulation and analysis, particularly when preparing datasets for model training or processing the results of SQL queries.
* **NumPy:** A foundational library for numerical operations, extensively used in conjunction with deep learning frameworks.

**Hardware Environment**

The hardware infrastructure is critical for handling the computational demands of training large language models and performing efficient inference for NL2SQL conversion.

**Development and Prototyping**

* **CPU:** A modern multi-core processor (e.g., Intel Core i7/i9 or AMD Ryzen 7/9) is sufficient for initial development, running smaller model experiments, and performing inference on less complex queries.
* **RAM:** A minimum of **16GB to 32GB** of RAM to comfortably handle datasets and load smaller pre-trained models.
* **Storage:** A **500GB+ Solid State Drive (SSD)** for fast loading of data, libraries, and model checkpoints, significantly improving development workflow efficiency.

**Cloud-Based Infrastructure (Scalable Deployment)**

For large-scale projects, production deployments, or when dedicated hardware is not feasible, cloud computing platforms offer flexible and scalable solutions.

* **Virtual Machines with GPUs:** Cloud providers like **AWS (Amazon Web Services), Google Cloud Platform (GCP), or Microsoft Azure** offer virtual machines pre-configured with powerful GPUs (e.g., NVIDIA Tesla V100, A100). This allows for on-demand access to high-end compute resources without significant upfront investment.
* **Managed Machine Learning Services:** Platforms such as **AWS SageMaker, Google Cloud AI Platform, or Azure Machine Learning** provide comprehensive environments for building, training, and deploying machine learning models, abstracting away much of the underlying infrastructure management.

**Exposure to New Tool**

Yes, during the course of this project, the team was introduced to and used **Bardeen** AI, a Chrome-based automation tool that allows users to create custom no-code web scrapers. This tool was particularly helpful for scraping structured travel destination data (name, description, tags, URLs) from the **Holidify** website. It simplified data collection without requiring manual Python scraping scripts like Selenium or BeautifulSoup, making it faster and more accessible

**Pseudocode for the Modules Implemented**

**Overall System Flow:**

Code snippet

FUNCTION NL2SQL\_System(natural\_language\_query, database\_schema):

parsed\_query = NaturalLanguageUnderstanding(natural\_language\_query, database\_schema)

sql\_query = SQLGeneration(parsed\_query, database\_schema)

results = DatabaseInteraction(sql\_query, database\_schema)

RETURN results

END FUNCTION

**Module 1: Natural Language Understanding (NLU) / Schema Linking**

This module aims to understand the user's intent and link entities in the natural language query to elements (tables, columns, values) in the database schema.

Code snippet

FUNCTION NaturalLanguageUnderstanding(natural\_language\_query, database\_schema):

// Tokenization and Preprocessing

tokens = Tokenize(natural\_language\_query)

processed\_tokens = Normalize(tokens) // Lowercasing, stemming, etc.

// Entity Recognition and Linking (often the most complex part)

linked\_entities = []

FOR each token IN processed\_tokens:

IF token matches a TABLE\_NAME in database\_schema:

Add {token, TYPE: TABLE, ID: table\_id} to linked\_entities

ELSE IF token matches a COLUMN\_NAME in database\_schema:

Add {token, TYPE: COLUMN, ID: column\_id, TABLE\_ID: parent\_table\_id} to linked\_entities

ELSE IF token represents a VALUE (e.g., numbers, dates, specific strings):

Add {token, TYPE: VALUE} to linked\_entities

ELSE IF token represents a SQL\_KEYWORD\_CONCEPT (e.g., "average", "count", "maximum"):

Add {token, TYPE: AGGREGATE\_FUNCTION, CONCEPT: "AVG" / "COUNT" / "MAX"} to linked\_entities

ELSE IF token represents a COMPARISON\_OPERATOR (e.g., "greater than", "less than"):

Add {token, TYPE: OPERATOR, CONCEPT: ">" / "<"} to linked\_entities

END FOR

// Determine Query Intent (e.g., SELECT, JOIN, GROUP BY, ORDER BY)

query\_intent = InferIntent(linked\_entities, processed\_tokens)

RETURN {linked\_entities, query\_intent}

END FUNCTION

**Module 2: SQL Generation**

This module takes the parsed intent and linked entities and constructs a valid SQL query.

Code snippet

FUNCTION SQLGeneration(parsed\_query\_info, database\_schema):

sql\_query\_parts = []

// Start with SELECT clause

SELECT\_CLAUSE = "SELECT "

IF parsed\_query\_info.query\_intent involves "all columns" OR no specific columns mentioned:

SELECT\_CLAUSE += "\*"

ELSE:

// Use linked columns, apply aggregates if inferred

FOR each entity IN parsed\_query\_info.linked\_entities:

IF entity.TYPE == COLUMN AND entity.CONCEPT == "AGGREGATE\_FUNCTION":

SELECT\_CLAUSE += entity.CONCEPT + "(" + database\_schema.get\_column\_name(entity.ID) + "), "

ELSE IF entity.TYPE == COLUMN:

SELECT\_CLAUSE += database\_schema.get\_column\_name(entity.ID) + ", "

SELECT\_CLAUSE = TrimTrailingComma(SELECT\_CLAUSE) // Remove last comma and space

Add SELECT\_CLAUSE to sql\_query\_parts

// FROM clause

FROM\_CLAUSE = "FROM "

principal\_table = IdentifyPrincipalTable(parsed\_query\_info.linked\_entities) // Determine primary table

FROM\_CLAUSE += database\_schema.get\_table\_name(principal\_table.ID)

Add FROM\_CLAUSE to sql\_query\_parts

// JOIN clause (if multiple tables are involved)

JOIN\_CLAUSE = ""

needed\_tables = GetRequiredTables(parsed\_query\_info.linked\_entities)

IF len(needed\_tables) > 1:

FOR each secondary\_table IN needed\_tables (excluding principal\_table):

join\_columns = FindJoinColumns(principal\_table, secondary\_table, database\_schema) // Based on foreign keys

JOIN\_CLAUSE += " JOIN " + database\_schema.get\_table\_name(secondary\_table.ID) + " ON " + \

database\_schema.get\_table\_name(principal\_table.ID) + "." + join\_columns.col1 + " = " + \

database\_schema.get\_table\_name(secondary\_table.ID) + "." + join\_columns.col2

Add JOIN\_CLAUSE to sql\_query\_parts

// WHERE clause (for filtering)

WHERE\_CLAUSE = ""

filters = ExtractFilters(parsed\_query\_info.linked\_entities) // e.g., "price > 100", "status = 'active'"

IF filters are present:

WHERE\_CLAUSE = " WHERE "

FOR each filter IN filters:

WHERE\_CLAUSE += database\_schema.get\_column\_name(filter.column\_id) + " " + filter.operator + " " + filter.value + " AND "

WHERE\_CLAUSE = TrimTrailingAND(WHERE\_CLAUSE)

Add WHERE\_CLAUSE to sql\_query\_parts

// GROUP BY clause (if aggregates are used)

GROUP\_BY\_CLAUSE = ""

grouped\_columns = IdentifyGroupByColumns(parsed\_query\_info.query\_intent, parsed\_query\_info.linked\_entities)

IF grouped\_columns are present:

GROUP\_BY\_CLAUSE = " GROUP BY " + JoinColumns(grouped\_columns)

Add GROUP\_BY\_CLAUSE to sql\_query\_parts

// ORDER BY clause (if sorting is implied)

ORDER\_BY\_CLAUSE = ""

order\_info = IdentifyOrderInfo(parsed\_query\_info.query\_intent, parsed\_query\_info.linked\_entities)

IF order\_info is present:

ORDER\_BY\_CLAUSE = " ORDER BY " + database\_schema.get\_column\_name(order\_info.column\_id) + " " + order\_info.direction

Add ORDER\_BY\_CLAUSE to sql\_query\_parts

// LIMIT clause (if limiting results)

LIMIT\_CLAUSE = ""

limit\_value = ExtractLimit(parsed\_query\_info.linked\_entities)

IF limit\_value is present:

LIMIT\_CLAUSE = " LIMIT " + limit\_value

Add LIMIT\_CLAUSE to sql\_query\_parts

RETURN Join(sql\_query\_parts, " ") // Combine all parts into a single SQL string

END FUNCTION

**Module 3: Database Interaction**

This module executes the generated SQL query against the database and fetches the results.

Code snippet

FUNCTION DatabaseInteraction(sql\_query, database\_schema):

TRY:

// Establish database connection

connection = ConnectToDatabase(database\_schema.connection\_string)

cursor = connection.cursor()

// Execute query

cursor.execute(sql\_query)

// Fetch results

results = cursor.fetchall()

// Get column names for results (optional, for better presentation)

column\_names = [description[0] for description in cursor.description]

// Close connection

cursor.close()

connection.close()

RETURN {data: results, columns: column\_names}

CATCH DatabaseError as e:

LogError("Database execution failed: " + e.message)

RETURN {error: "Failed to execute query"} END FUNCTION

**Experimental Results**

**Result Display**

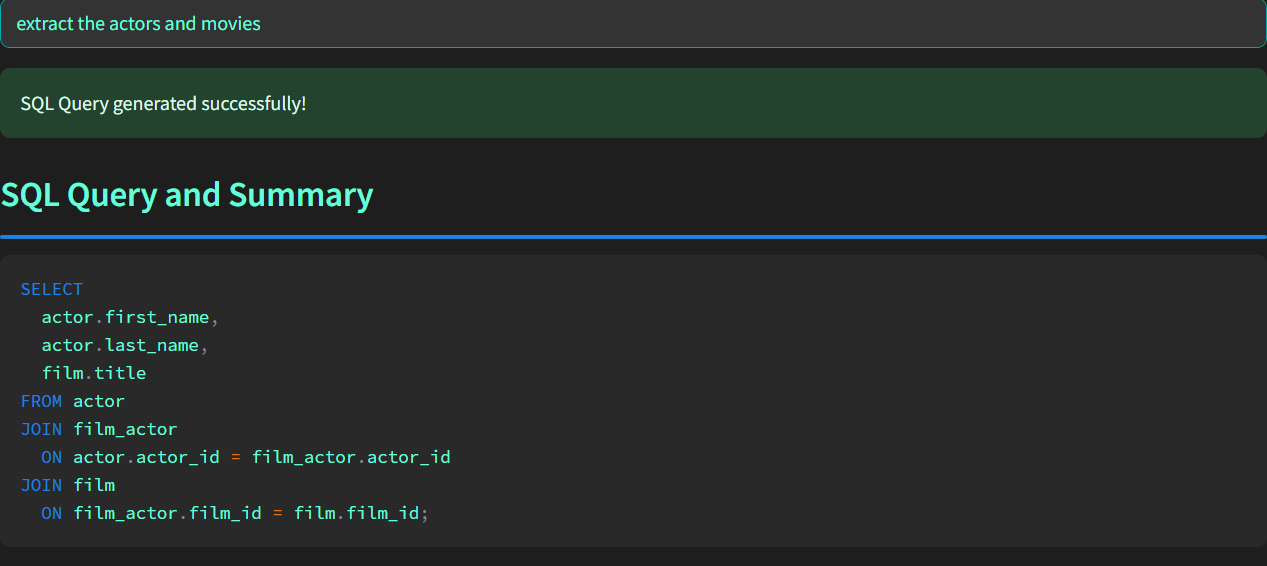
The application was tested with a variety of queries such as:

* "Extract the customers and products purchased"
* "Extract the Actors and their Movies"
* "Extract the highest paid Actor"
* “Extract the Monthly sales”
* “Extract the DVD’s rented by customers”
* “Insert a customer and movie rented by him”
* “Extract the Monthly Orders and Sales Metrics”

**Comparison Between Existing Travel Platforms and Proposed System**

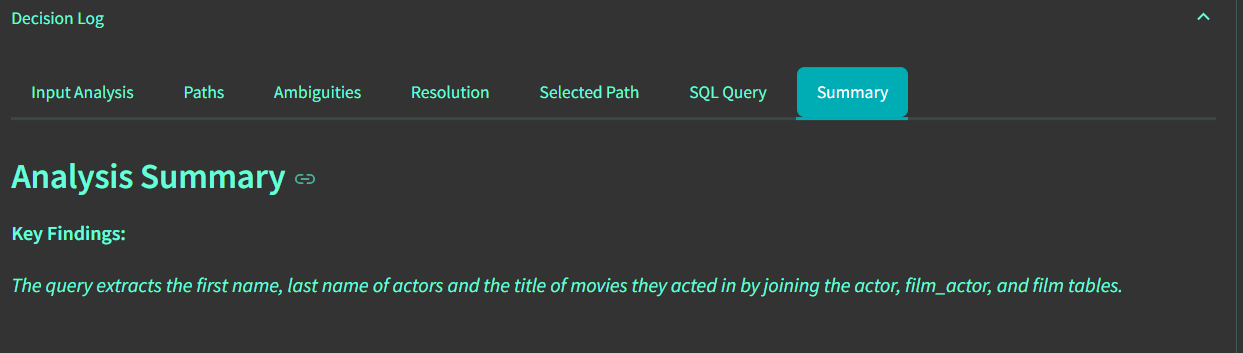
|  |  |  |
| --- | --- | --- |
| Feature | Existing technologies | This Project (Proposed) |
| Free-text Query Support | Limited | Fully Supported |
| SQL Complexities | Limited to predefined patterns; struggles with joins, aggregations. | Excellent; robustly handles complex SQL constructs. |
| Database Adaptability | Requires significant effort to adapt to new schemas/domains. | Highly adaptable; supports various RDBMS; efficient schema integration. |
| Query Interpretation | Keyword-based | Semantic with NER + Tokenization |
| Ambiguity Handling | Poor; often requires explicit disambiguation. | Good; incorporates mechanisms for clarifying ambiguous queries. |
| Real-time Performance | Fast for simple queries. | Optimized for real-time performance, balanced with accuracy. |
| Query Analytics | None inherently. | Core Feature: Provides detailed analytics on user queries and system performance. |

1)



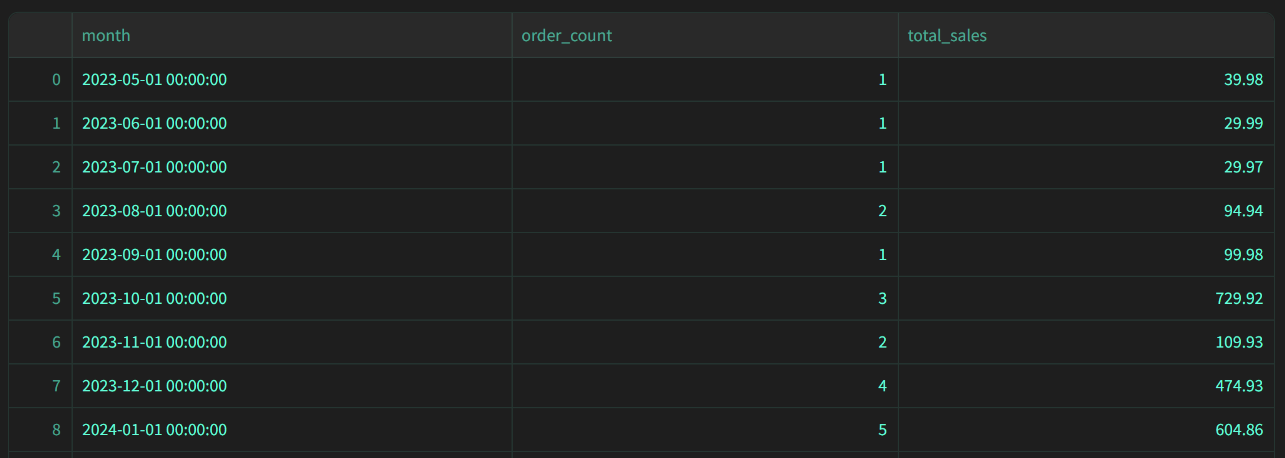
The image shows a user inputting the natural language query "extract the actors and movies", followed by a successful conversion message and the automatically generated SQL query. The SQL query correctly identifies relevant tables (actor, film, film\_actor), selects the appropriate columns (actor.first\_name, actor.last\_name, film.title), and performs the necessary multi-table joins to fulfill the request, highlighting the system's ability to understand schema relationships and complex queries. The clear UI indicates a user-friendly system.

2) Key findings related to Query

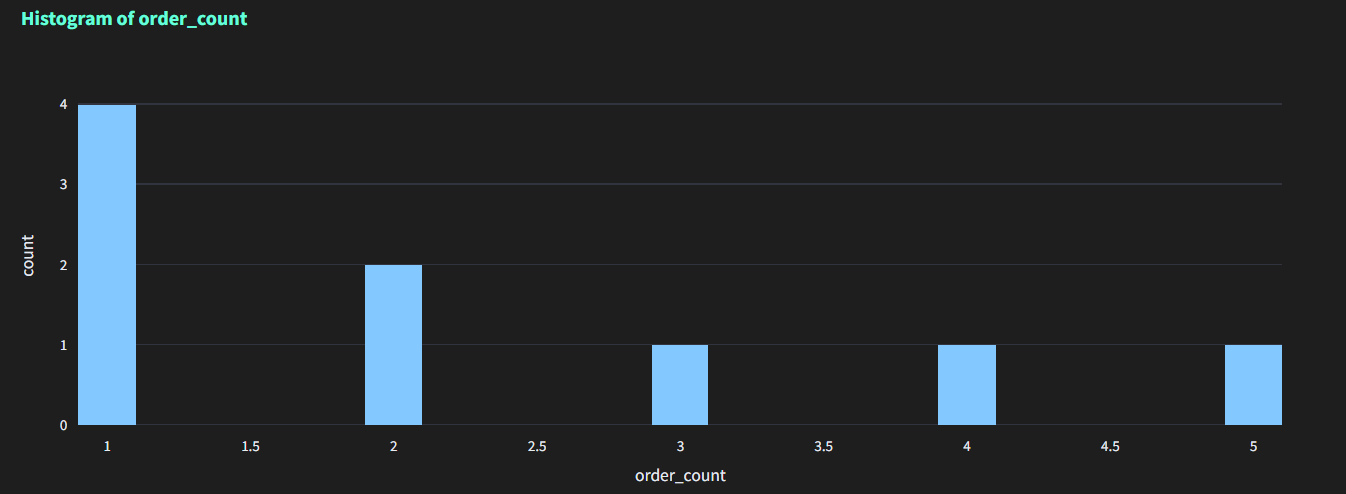


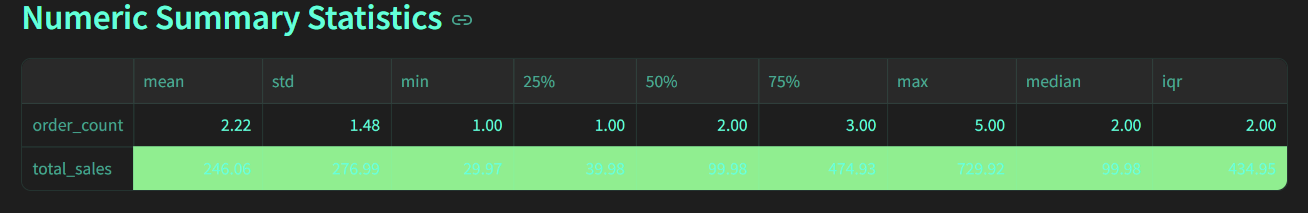
3)

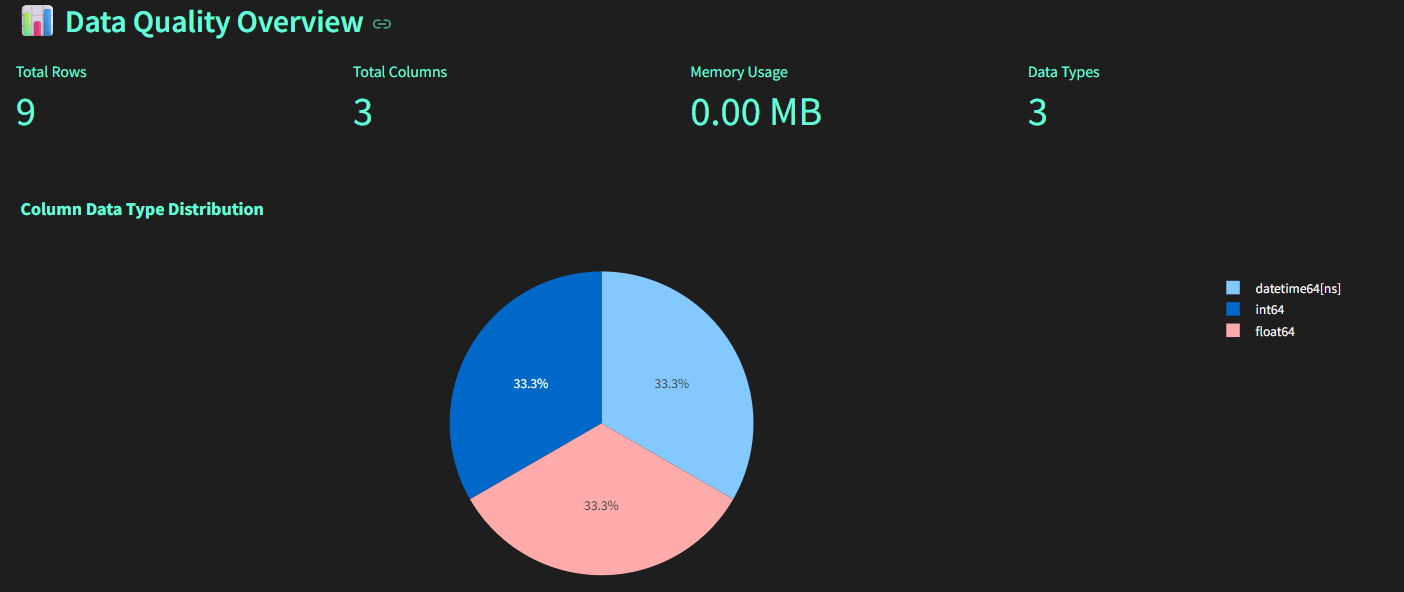




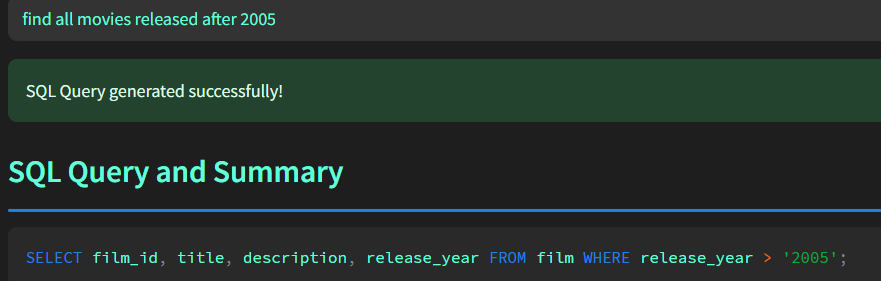
4)Analytics related to above Query





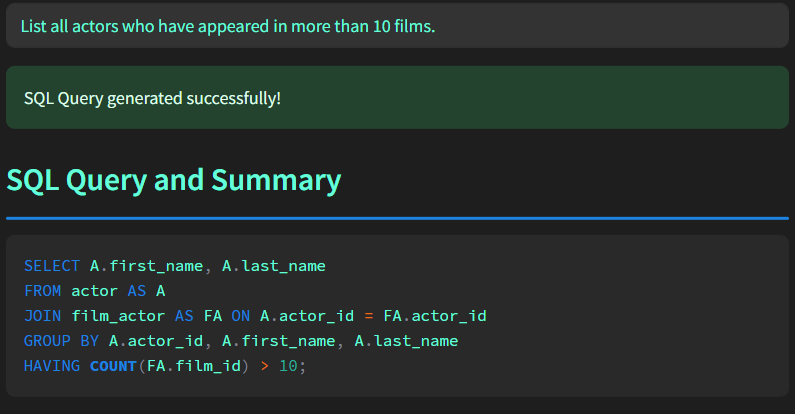


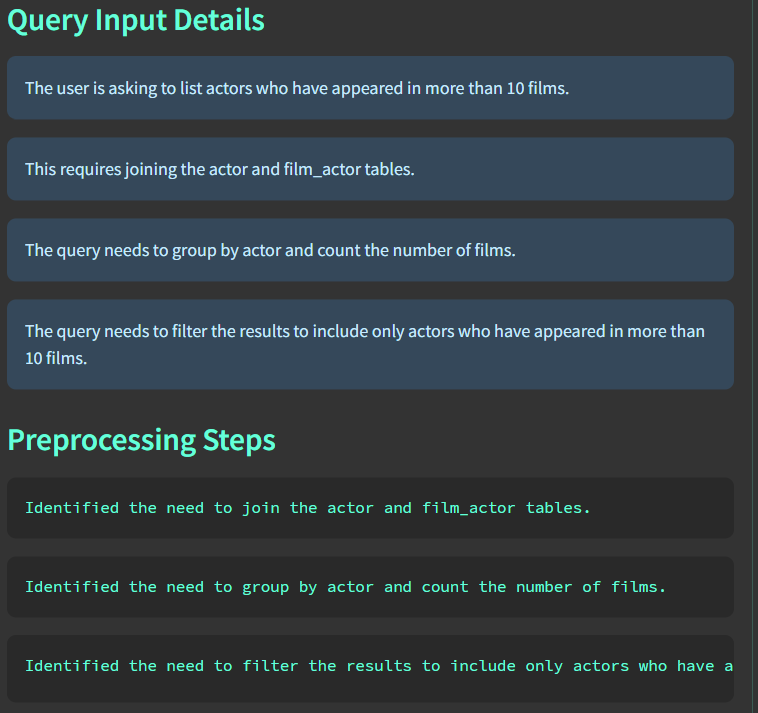
5)Queries for Sakila Database



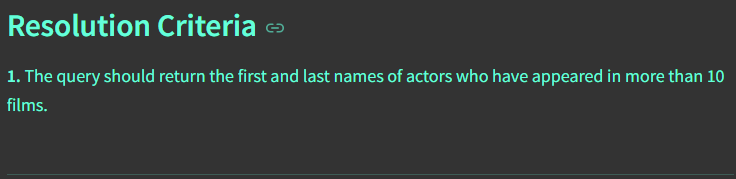


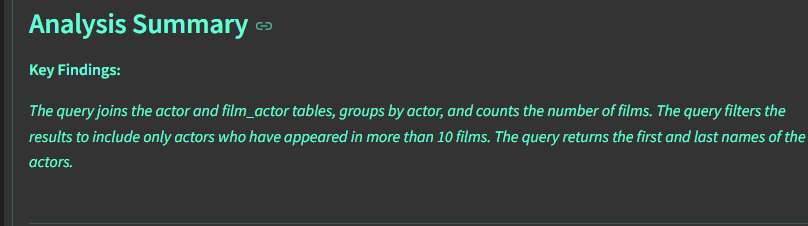
6)Decision Log Analysis for Query “List all actors who have appeared in more than 10 films.”

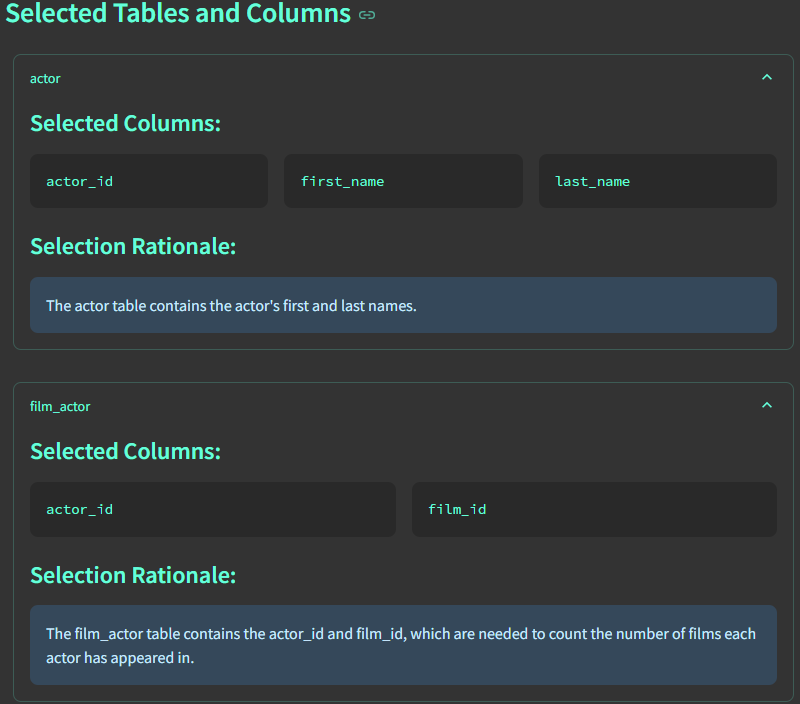












**Conclusion**

In conclusion, this NL2SQL project successfully developed a robust system capable of translating natural language queries into accurate and executable SQL, demonstrating proficiency in handling complex database schemas and multi-table relationships. Beyond core translation, the integration of comprehensive query analytics provides a critical feedback loop, offering invaluable insights into user behavior, query patterns, and system performance, thereby not only democratizing data access but also empowering continuous improvement and strategic data governance. This dual capability positions the tool as a significant advancement in facilitating intuitive and intelligent data interaction.