31 october 2025 Deadline Report

Francesco Pivotto 2158296
Giorgia Amato 2159999
Alessio Demo 2142885

27 ottobre 2025

Indice

1 Introduction

In this part of our project, we deal with the database containing all the data that we need for our purpose. In details we need to establish a connection to connect the database with the data resources, that allows us to create an "running instance" of the database which we can use to query the database itself and store the results returned, in particular in our case the results will be stored in a custom JSON format called 'Simple JSON format' provided by the tutor. Next we will carry out the evaluation of the results produced by our system using the ground truth data as comparator.
Let's go into details.

2 Libraries & Components

2.1 System Architecture: Dependency Stack and Key Requirements:

The project is built upon a Python dependency stack prioritizing robustness, efficient data analysis, and flexible configuration. All essential libraries are listed in requirements.txt.

1. Core Frameworks & Data Analysis (DataFrames): **pandas** and **numpy** are used for the efficient manipulation and analysis of tabular data structures (DataFrames) and complex numerical calculations.

2. Database and SQL: **duckdb**: Utilized as a high-performance, in-process analytical database, ideal for rapid processing of data volumes directly within the application. **sqlglot**: Employed for the abstract analysis, manipulation, and translation of SQL queries.

3. LLM Orchestrator (Large Language Model) **LangChain**: The core framework managing interactions with AI models. **langchain-google-genai**: Provides access to Google Gemini AI models. This setup ensures modularity, flexibility, and scalability in establishing the system's communication with the LLM.

4. Configuration and Data Validation: **pydantic**: Essential for data validation and structured configuration management. It enforces type safety, handles data parsing/coercion, defines structured nested models, and raises immediate errors on malformed input. **python-dotenv**: Loads environment variables (including secrets) from the .env file. **pyyaml**: Handles reading and writing configuration files in YAML format.

5. Templating: **jinja2**: The templating engine used for the dynamic generation of structured output, such as code, reports (HTML, XML, JSON), or complex prompt statements, by separating presentation logic from data.

6. Logging and Utilities **loguru**: The chosen logging library for its simplicity and structured output, which enhances system log readability. Logs: All runtime logs are stored in the .log folder.

7. Development and Testing These dependencies are specific to the development environment and the test suite: **pytest**: The standard testing framework for executing

all tests. coverage and **pytest-cov**: Used to measure the percentage of source code covered by automated tests (Code Coverage). **pytest-mock**: Facilitates the creation of mock objects and stubs to isolate external dependencies during unit tests.

3 Repository Structure

The Structure of out project is organized in scopes (data resources that include database data - ground truth data and output results, configuration stuff, source scripts, tests), so we have:

- **data** folder: Contains all the datasets with the ingest, queries and table creation files, in this folder we store also the ground truth data and the results delivered by our system.
- **config** folder: All the scripts and config files for the initialization of the system are there.
- **src** folder: Here we have the scripts that develop the core functionalities of our system and also other useful utils.
- **test** folder: Finally this folder contains the unit tests that check the system correctness.

4 Implementation Details

4.1 Configuration handling

All the system configuration details are reported in **.yaml** file, and the **src/settings.py** is the central hub for all configuration data. It handles the critical process of loading and validating the external YAML file, ensuring the application always runs with correct and predictable parameters. Going more deeply:

- Parsing (YAML): The script uses **pyyaml** to read and parse the content of the config.yaml file into a standard Python dictionary.
- Structured Validation (Pydantic): The parsed data is immediately processed and validated using Pydantic.
  - Structure: We define precise, nested classes (like ModelConfig, IOConfig, ExecutionConfig, etc.) that mirror the expected structure of the YAML file. The main class, AppConfig, aggregates all these sections.
  - Type Safety: Pydantic strictly enforces data types (e.g., ensuring temperature is a float, and max_retries is an int). This eliminates common runtime errors caused by misconfigured settings.
  - Predictability: By validating the structure and types upfront, we guarantee that all required fields are present and correctly formatted, making the configuration predictable and reliable.

- Global Access: The configuration is loaded into a single, global instance called settings. This allows any module in the project to access any configuration detail easily and consistently, such as settings.gemini.temperature.

4.2 Database-Dataset Connection and Management

For our purpose we will use DuckDB as database manager, we believe that it is the ideal for our project since it is very powerful and rapid in the data processing within the system, moreover duckDB is an embedded database manager and can run within the instance of our system instead of establishing a server connection like postgres. At more technical level the **db_connection.py** connects to a DuckDB database file associated with a specific dataset. The next step is **duckdb_db_graphdb.py** which allows us to automatically build DuckDB databases from a directory of datasets, each containing an SQL ingestion script that creates the data tables and populates them, this allows us to have an instance of the db representing a particular dataset in which we can run queries.

4.3 Query Execution

The queries can be execute by means **run_queries_to_json.py** script that automates the execution of SQL queries on DuckDB databases and exports their results to JSON files, one per query.
Going more deep the script executes each query and saves the results as JSON:

1. Iterates through each query tuple.
2. Executes the SQL query using the provided DuckDB connection.
3. Fetches the resulting rows and column names.
4. Converts the result into a list of dictionaries.
5. Saves each query result as a JSON file in the output directory.

4.3.1 Explain Plans (DuckDB)

The `EXPLAIN` module was implemented to validate the SQL execution pipeline on DuckDB and to visualise the difference between logical and physical plans, providing a reliable baseline for the comparison with the GALOIS framework.

Implementation Overview The functionality is implemented in the main script `src/db/run_explain_plans.py`, which delegates the actual DuckDB interaction to the adapter `src/db/duckdb\_explain.py`. The script can be executed by passing either a specific dataset name (e.g., `world`) or the keyword `all`. For each dataset located in `<repo-root>/data/<dataset>`, the corresponding database `<dataset>.duckdb` and all `queries_*.sql` files are loaded, split into single statements, and both `EXPLAIN` and `EXPLAIN ANALYZE` are executed.

The module's core responsibilities are:

1. **Dataset iteration:** enumerates datasets and SQL files and assigns a stable query identifier (e.g., `queries\_world\_\_q3`);

2. **Parsing:** splits multi-statement SQL files on semicolons while preserving comments for traceability;

3. **Execution:** connects to the dataset's `.duckdb` database and issues both `EXPLAIN` and `EXPLAIN ANALYZE`;

4. **Persistence:** stores all outputs through the `save_both()` function.

Output Structure The results are written under the global `results/` directory in two parallel trees:

- `explain_result/` for text files (`.txt`) for human inspection;
- `explain_result_json/` equivalent JSON files for machine processing.

Each of these folders contains one subdirectory per dataset (`flight-2`, `flight-4`, `fortune`, `geo`, `premier`, `presidents`, `world`). For every SQL statement, four output files are produced sharing the same base name (e.g., `queries_world__q3`):

- `__explain.txt` and `__explain.json` for logical and physical plans;
- `__analyze.txt` and `__analyze.json` for runtime statistics and operator timings.

Example A representative example from the `world` dataset is:

```
SELECT count(DISTINCT government_form)
FROM target.country
WHERE continent = 'Africa';
```

This query counts the number of distinct government forms for African countries. The generated logical plan follows a typical structure ($Aggregate \rightarrow Filter \rightarrow TableScan$), while the physical plan contains the corresponding operators ($HASH\_AGGREGATE \rightarrow FILTER \rightarrow SEQ\_SCAN$). The analyzed output reports the number of processed rows and the execution time per operator.

Role in the Project This component ensures that the database layer behaves as expected and that query plans can be reliably generated and compared. The outputs will later be used to align relational operators with their LLM-based equivalents within the GALOIS pipeline.

4.4 Evaluation

We have the **galois_eval.py** provided by Tutor, that is a command-line evaluator that computes performance metrics for query results across different datasets. Briefly it standardizes different result formats, normalizes textual/numeric data, computes multiple precision–recall-based metrics, aggregates them across datasets, and outputs results in several human- or machine-readable formats.

4.5 Statistics for the datasets in the experiments

Table 1 provides an overview of the datasets used our project. The datasets are divided into two main categories based on the type of task: IK (Incomplete Knowledge) and MC (Multiple Choice). The MC-type datasets include PREMIER, sourced from BBC, and FORTUNE, based on Kaggle data. Finally, GEO-TEST represents the test dataset, also derived from Spider [68], used to validate the model's performance.

Dataset name	**Dataset source**	**# of queries**	**Avg. expected cells**	**Type**
FLIGHT2	Spider [68]	3	1.0	IK
FLIGHT4	Spider [68]	3	534.0	IK
FORTUNE	Kaggle	10	11.1	MC
GEO	Spider [68]	32	23.1	IK
MOVIES	IMDB	9	50.9	IK
PREMIER	BBC	5	57.4	MC
PRESIDENTS	Wiki	26	41.2	IK
WORLD	Spider [68]	4	33.2	IK

Tabella 1: Description of used datasets

For the **Avg_expected_cells** parameter we have implemented an ad hoc script that calculates it, more technically the **avg_cells_metric.py** calculates for each dataset the sum of the cells returned by the results of each query and then adds them together. Finally, it calculates the metric by dividing the total number of cells by the number of queries for that specific dataset.