UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

UNIVERSITY OF PADUA
DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN COMPUTER ENGINEERING

"REENGINEERING THE GALOIS SYSTEM TOWARD REPRODUCIBLE AND GENERALIZABLE LLM-BASED STRUCTURED RETRIEVAL"

Supervisor: Prof. GIANMARIA SILVELLO
Co-supervisor: Dott. MIRCO CAZZARO

Candidate: FRANCESCO CHEMELLO
Student ID: 2121346

ACADEMIC YEAR 2025 – 2026

Graduation date: April 13th, 2026

Sommario

In questa tesi viene presentato il sistema GaloisPy, uno strumento software scritto in Python che re implementa il sistema Galois, un'applicazione Java sviluppata da un gruppo di ricercatori dell'Università della Basilicata in collaborazione con l'istituto di ricerca francese EURECOM per il recupero di dati strutturati dai modelli linguistici di grandi dimensioni, o LLM, utilizzando la sintassi SQL. GaloisPy introduce anche diverse migliorie ed ottimizzazioni mirate a migliorare le prestazioni del sistema come la gestione della memoria della chat, la creazione dinamica di modelli Pydantic e la riscrittura dei prompt dalla sintassi SQL a linguaggio naturale, quest'ultimo rivelatosi fondamentale per l'accuratezza del sistema. In questo lavoro vengono presentati i risultati dei diversi esperimenti condotti confrontando i valori ottenuti dal sistema GaloisPy e dal sistema Galois utilizzando il modello *GPT-4.1 nano* di OpenAI. Questi test analizzano le prestazioni di ambedue i sistemi sia in un contesto in cui le risposte vengono generate utilizzando l'intelligenza intrinseca (*Internal Knowledge* or IK) dei modelli LLM, sia utilizzando strategie di *Retrieval-Augmented Generation* (RAG). Dai risultati ottenuti, emerge che il sistema GaloisPy ha ottenuto una qualità dei dati generati nettamente superiore rispetto all'originale, come attestato dalle diverse metriche utilizzate. Vengono tuttavia evidenziate alcune sue limitazioni significative, come l'inefficacia della strategia di *Key-Scan* rispetto alla *Table-Scan* e la difficoltà nella gestione di query complesse con clausole *JOIN*. Infine, nel contesto RAG, emerge come l'uso di modelli di reranking e le ridotte dimensioni del modello LLM possano penalizzare le performance complessive.

Abstract

In this thesis, we present the GaloisPy system, a software tool written in Python that re-implements the Galois system, a Java application developed by a team of researchers from the University of Basilicata in collaboration with the French research institute EURECOM for retrieving structured data from large language models, or LLMs, using SQL syntax. GaloisPy introduces several improvements and optimizations designed specifically to enhance its performance, such as including chat history management, dynamic Pydantic models for structured outputs, and prompt rewriting from SQL-like language to natural language, with the last being crucial for the system's accuracy. In this work, the results of various experiments conducted will be presented, comparing the performance of GaloisPy against the original Galois system using the *GPT-4.1 nano* model by OpenAI. These tests analyze both systems' performance in different scenarios, including one where data is generated solely using LLM internal knowledge and another employing the *Retrieval-Augmented Generation*, or RAG, strategy. From the results obtained, it is evident that GaloisPy achieves significantly better performance compared to the original, as indicated by the metrics used. However, it also has some limitations, such as the ineffectiveness of the *Key-Scan* strategy compared to the *Table-Scan*, and challenges in managing complex queries, especially those involving JOIN clauses. Finally, in the RAG scenario, it was observed that using a reranking model, coupled with the small size of the LLM used, can reduce overall performance.

Index

Introduction ..... **I**

Chapter 1: The Galois system **1**

1.1 - Overview 1

1.2 - Architecture and workflow 3

1.3 - Logical and physical optimizations 6

1.4 - Limitations 9

Chapter 2: GaloisPy, the Galois system re-implemented in Python **11**

2.1 - Overview 11

2.2 - Architectural patterns and design patterns 12

2.3 - Architecture 14

2.3.1 - Base structure 14

2.3.2 - Components interaction and workflow 16

2.4 - Optimizations 20

2.5 - Distribution and usage 24

Chapter 3: Internal knowledge experiments **27**

3.1 - Experimental setup 27

3.2 - Evaluation metrics 29

3.3 - Results and comparisons 31

3.3.1 - Galois vs GaloisPy 32

3.3.2 - Performance analysis of the different GaloisPy configurations 38

Chapter 4: Retrieval-augmented generation experiments 47

4.1 - Experimental setup 47

4.2 - Evaluation metrics 49

4.3 - Results and comparisons 49

4.3.1 - Galois vs GaloisPy 49

4.3.2 - Performance analysis of the different GaloisPy configurations 50

Chapter 5: The Nobel Prize experiment 57

5.1 - Experimental setup 57

5.2 - Results and comparisons 58

5.2.1 - Template number 1 58

5.2.2 - Template number 2 62

5.2.3 - Template number 3 65

5.2.4 - Template number 4 68

5.2.5 - Template number 5 71

5.3 - Conclusions 74

Conclusion and future work 75

Appendix A: Large language models 79

A.1 - LLMs and transformers 79

A.2 - Generation process 81

Appendix B: Databases 83

B.1 - Relational databases 83

B.2 - SQL syntax and queries 84

Bibliography and webography I

Introduction

The integration between large language models, or LLMs for short, and structured data retrieval has become a pivotal area of research in data retrieval. LLMs, during their pretraining, learn a huge amount of general knowledge, but due to their probabilistic nature, they are challenging to use as a reliable source of knowledge.

In June 2025, a team of researchers from the University of Basilicata, in collaboration with the French research center EURECOM, published a paper titled *"Logical and Physical Optimizations for SQL Query Execution over Large Language Models"*, in which they present Galois, a Java application that combines the vast knowledge of large language models with the power of SQL syntax to retrieve structured data, mimicking the behavior of a relational database management system (RDBMS). The core functionality of Galois is the scan operators, which are designed to interact with the LLM. These novel operators, named *Key-Scan* and *Table-Scan*, iteratively query the AI model, and the difference between these two is in how they retrieve data. The first one asks the model to furnish all the primary key values regarding the query to later use them as a hint for retrieving the full tuple, while the second one asks the model to provide all the attributes involved in the query. Both operators can dynamically include in the request some or all the relevant filtering conditions to narrow the result set generated by the LLM. This optimization, together with the choice of the scan operator, is managed by the LLM itself, thanks to two optimizations, named respectively the *logical optimization* and the *physical optimization*.

While the results obtained with Galois were promising, this system presents some limitations, the main one being the architectural complexity. Due to its implementation in Java, Galois features a complex and non-intuitive architecture that makes it difficult to use and, more importantly, to extend. To address these issues, in this thesis, we are going to present GaloisPy, a Python software tool that re-implements the same functionalities of Galois, providing a simpler, cleaner, and easier-to-use implementation, leveraging some of the most popular and widely used libraries such as *Langchain*, *Llama-index*, and *Pydantic*. Nevertheless, GaloisPy is not just a mere modern re-implementation of Galois

in Python, but it also introduces several optimizations and architectural enhancements, including chat history management, dynamic Pydantic models for structured outputs, and prompt rewriting from SQL-like to natural language. The whole software is distributed as a *wheel* package instead of a standalone application like Galois, to be easily integrated into any Python project, and the codebase, together with the experimental results, is available on GitHub[1].

To evaluate the capabilities of our system, we have conducted several tests using the same test suite used in the Galois original paper. These tests cover different topics and different query structures, using two sources of knowledge: one relies only on the LLM's internal knowledge, indicated as *Internal Knowledge* (IK), while the other adopts the *Retrieval-Augmented Generation* (RAG) pipeline. In addition to these, we have developed a novel experiment using an additional dataset about the Nobel Prize awardees and winners, in which we test the capabilities of the GaloisPy systems over different query templates. The novelty of this experiment is that we analyze how the system reacts to the different queries in which only the filtering clause, which corresponds to the WHERE clause in SQL, is changed. For these experiments, we have used the *GPT-4.1 nano* model from OpenAI. Different from what the Galois team did, we have decided to test the capabilities of our system using one of the smallest models available nowadays. Doing so, we aim to prove the effectiveness of our re-implementation even in scenarios where users do not have access to a powerful and yet expensive model, such as *Llama 3.1 70b*, the LLM model the Galois team uses for their experiments. On the other hand, in the case of RAG experiments, we have opted to use the *mxbai-embed-large* as the embedding model and the *mxbai-rerank-xsmall-v1* as the reranking model. Both of them are very small, but in this case, the choice was limited to these two due to hardware limitations rather than a specific design choice, since both systems are run locally.

From the experimental results, it emerged that GaloisPy is superior to Galois with the same model and with the same configuration; however, when the novel improvements are used, the GaloisPy performance increases a lot, especially with the usage of the prompt rewrite. Indeed, in some cases, it equaled the scores obtained by the Galois system with far more powerful LLMs model, such as the *GPT-4o mini*, *Llama 3.1 8b*, and *Llama 3.1*

[1] Repository link: https://github.com/FrancescoChemello/Galoispy/.

70b. On the other hand, the results obtained by the RAG test suite are decent but lower than the ones achieved on the IK experiments, highlighting the limitations of such a modality with small LLM models. In addition, the introduction of the reranking model has worsened the overall performance in almost all the scenarios tested, highlighting the limitations of such a model. Regarding the Nobel Prize experiment, GaloisPy has performed quite well on queries without a JOIN condition, but really poorly on queries with it, showing some difficulties in the management of join operations. Lastly, different from what is reported in the original paper, the *Key-Scan* retrieval mode has achieved far poorer performance compared to the *Table-Scan* one, suggesting that using a simpler and more complete prompt can effectively increase the quality of the results generated.

The outline of this thesis is the following: *Chapter 1* introduces the original Galois system, covering all relevant components of its implementation. *Chapter 2* presents our software tool, GaloisPy, detailing its architecture, features, and the optimizations introduced. *Chapters 3 and 4* describe the experimental setup and results for *Internal Knowledge* (IK) and *Retrieval-Augmented Generation* (RAG) scenarios, respectively. *Chapter 5* presents the novel Nobel Prize experiment, analyzing how the system handles different query templates. Finally, the last section draws conclusions and suggests directions for future work.

In conclusion, GaloisPy provides a simpler, clearer, and more reliable solution compared to the original system, Galois, offering higher performance given the same LLM model; however, it presents some limitations, such as the ineffectiveness of the *Key-Scan* strategy and the poor management of JOIN conditions, both of which are inherited from the original implementation. We believe that overcoming these limitations by providing a new retrieval strategy or by improving the existing *Table-Scan* operator, coupled with better management of the JOIN conditions, will make GaloisPy a reliable tool for structured data retrieval. At the same time, we cannot dismiss the idea of modifying such a tool into a more sophisticated one by integrating an always up-to-date source of knowledge, or by evolving it into a *Model Context Protocol*, or MCP for short.

Chapter 1: The Galois system

SIGMOD is the *Association for Computing Machinery's Special Interest Group on Management of Data*, which specializes in large-scale data management problems and databases. Every year, since 1975, the *ACM SIGMOD Conference* has taken place, in which researchers from all over the world participate, presenting their research in the field of information retrieval, or for short, IR. Important database publications have appeared first at *SIGMOD,* like, for instance, the *PostgreSQL* database system, presented by its creator Michael Stonebraker in 1986, the very first "next generation" RDBMS that is still today one of the most used software for managing relational databases.

In June 2025, a group of researchers from the University of Basilicata and from the EURECOM research institute presented at ACM SIGMOD Conference 2025 their research paper titled *"Logical and Physical Optimizations for SQL Query Execution over Large Language Models"*, in which they define a new database system, called Galois, in honor of the famous mathematician and political activist Évariste Galois, founder of the Galois theory and the group theory, two major branches of abstract algebra, that is a Java software that combines SQL with AI models to retrieve reliable and structured data from the vast internal knowledge of LLMs. The project is presented as an adapter between these two, allowing the user to define their own SQL queries over its own relational database, and receiving as an output a file containing the query result, as a traditional RDBMS does.

In this chapter, we are going to present how the project Galois was developed and what its main components and functionalities are.

1.1 - Overview

Galois is a novel database system that sits between a query and an LLM that aims to enhance SQL results by applying query optimizations specifically designed for LLMs. They have designed alternative physical operators for retrieving information, while they have adapted traditional optimization strategies to this innovative approach. Their work

has also been focused on the trade-off between quality and execution cost in terms of token consumptions, providing different scan strategies, like, for instance, one that pushes down conditions during the retrieval process, which reduces the number of calls to the model while maintaining discrete performance in terms of query result quality.

While traditional extraction methods often require manual and complex parsing processes or complicated pipelines to retrieve structured data from LLMs, this novel database system proposes a solution that overcomes both those issues, providing a valuable tool for interpreting and extracting data from extensive text corpora encapsulated within the neural network, while maintaining an easy-to-use interface.

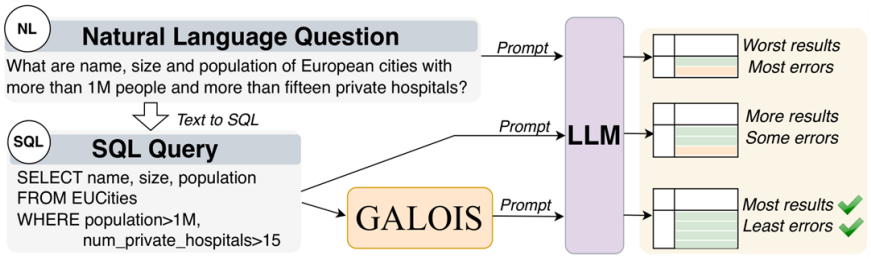


Figure 1 - Comparison between traditional retrieving strategies and Galois.

Using Galois is quite simple. Users should download the software from the GitHub repository, configure their local PostgreSQL database, define a query file, and invoke the shell script to execute it.

galois_query.json

```
{
  "databaseDriver": "org.postgresql.Driver",
  "databaseURI": "jdbc:postgresql:llm_directors_movies",
  "databaseSchema": "public",
  "databaseUser": "pguser",
  "databasePassword": "pguser",
  "sql": "SELECT m.originaltitle FROM movie m WHERE m.director='Steven Spielberg'",
  "confidenceThreshold": 0.6
}
```

Figure 2 - Example of user-defined query for Galois [source GitHub official repository].

1.2 - Architecture and workflow

Galois presents a rather complex architecture, but its core can be divided into two main components:

1. The *logical component* that decomposes an SQL query into a series of logical steps, representing the execution pipeline.
2. The *physical component* that interacts with the large language model to retrieve raw data.

Starting with the *logical component*, this module receives as input the user-defined SQL query and then generates as output an execution plan composed of a series of logical operators, such as *Selection*, *Projection*, and *Join*, that work similarly to their namesakes in a relational database. They process the raw data received from the LLM and manipulate it appropriately according to each logical operator, like, for instance, the *Selection* operator applies a filtering condition to the data retrieved. These logical operators are arranged in a pipeline, where the output of one operator serves as the input for the next.

Additionally, to interact with the LLM, Galois introduces two novel operators: the *LLMScan* and the *Filter-LLMScan*. The key difference between them resides in their use of filtering conditions: the first operator does not include any filtering conditions in the retrieval prompts, while the other incorporates some to narrow the data criteria. Deciding whether to use *LLMScan* or *Filter-LLMScan* is delegated to the *logical optimizer*, a component that will be presented later.

Operator	Symbol	Description
LLMScan	$\mathcal{S}$(LLM)	Fetch data from LLM
Filter-LLMScan	$\mathcal{S}_{cond}$(LLM)	Fetch data from LLM w.r.t. cond
Selection	σ_{cond}	Select tuples w.r.t. cond
Projection	π_{attrs}	Extract attrs from tuples
Join	$\bowtie_{cond}$	Join two table given cond
Distinct	δ	Removes duplicate tuples
Grouping	γ_f	Groups tuples on common values and compute f over groups

Figure 3 - Logical operators supported by Galois.

Moving to the next module, the *physical component*, its responsibility is to query iteratively the LLM model to extract the tuples. It is constituted of two retrieval strategies: *Table-Scan* and *Key-Scan*.

On the one hand, the *Table-Scan* strategy is the most straightforward one, and it consists of querying the LLM using an SQL-like prompt that involves all the attributes required in the SELECT clause. This approach aims to reduce the number of iterations and, consequently, the number of tokens used by requesting all the information in a single question.

Algorithm 1: Table-Scan

Input: SQL query q, table t_{name} , db schema s, max iter. $maxIter$, language model LLM
Output: tuple set t

```
1  i = 0; prompt = ""; context=[]; t = { };
2  while i < maxITer do
3      if i == 0 then
4          prompt = genFirstPrompt(t_name, s, q);
5      else
6          prompt = genIterativePrompt();
7      jsonResponse = LLM.request(prompt, context);
8      parsedTuples = parse(jsonResponse, t_name, s);
9      if noNewTuples(parsedTuples, t) then
10         break;
11     else
12         context.add(prompt);
13         context.add(jsonResponse);
14         t.addAll(parsedTuples);
15     i++;
16 return t;
```

Figure 4 - Table-Scan algorithm.

On the other hand, the *Key-Scan* approach utilizes a two-step strategy, implementing a sort of *Chain-of-Thoughts*. In the first phase, Galois asks to retrieve all the primary key values, then, in the second phase, it iteratively asks the model to fill the other tuples' fields by passing the value of a primary key previously retrieved as a hint. This strategy, different from the previous one, aims to improve the overall quality of the result set by asking simpler but more specific questions. However, *Key-Scan*, as we will see later, significantly increases the total number of iterations and, thus, the total number of tokens.

Algorithm 2: Key-Scan

Input: SQL query q, table t_{name} , db schema s, max iter. $maxIter$, language model LLM
Output: tuple set t

```
1  i = 0; prompt = ""; attrKeys=t_name.keys;
2  context=[]; keys= { }; t = { };
3  while i < maxITer do
4      if i == 0 then
5          prompt = genFirstPromptKey(t_name,s,attrKeys,q);
6      else
7          prompt = genIterativePromptKey();
8      jsonResponse = LLM.request(prompt, context);
9      parsedKeys = parseKeys(jsonResponse, t_name, s);
10     if noNewKeys(parsedKeys, keys) then
11         break;
12     else
13         context.add(prompt);
14         context.add(jsonResponse);
15         keys.addAll(parsedKeys);
16     i++;
17 noKeysAttrs = t_name.attrs - t_name.keys;
18 for kVal ∈ keys do
19     promptT = genTuplePrompt(noKeysAttrs, kVal, s);
20     jsonResponse = LLM.request(promptT);
21     parsedTuple = parse(jsonResponse, t_name, s);
22     t.add(parsedTuple);
23 return t;
```

Figure 5 - Key-Scan algorithm.

To obtain a structured JSON response, an output schema is passed within the question, along with the database schema, also in JSON format, to provide full context to the model. In case of an incomplete JSON response, Galois tries to parse the output to capture needed information and, if it succeeds in such a task, it reconstructs the expected tuple, without wasting the LLM's generated output.

Also in this case, the choice between *Table-Scan* and *Key-Scan* is delegated to the *physical optimizer*, a component that will be presented later.

Finally, let us understand how Galois operates and what its workflow looks like. First, Galois receives from the user a SQL query together with the parameters for the execution and the credentials to access the local database. Then it defines the logical plan by composing the query into a logical pipeline, using the operators available. Then it executes the pipeline one step at a time, starting from the scan operators such as *LLMScan* and *Filter-LLMScan*. Finally, it returns to the user the result set in the form of a CSV file.

1.3 - Logical and physical optimizations

For the same query, multiple correct logical plans can be created, so how does Galois decide which logical plan to use? Indeed, Galois adjusts its execution plan by changing the moments at which to apply a particular filtering condition or whether to use the *Key-Scan* or the *Table-Scan* strategy.

All these variations to the original plan are driven by two optimizers: *the logical optimizer* and *the physical optimizer*.

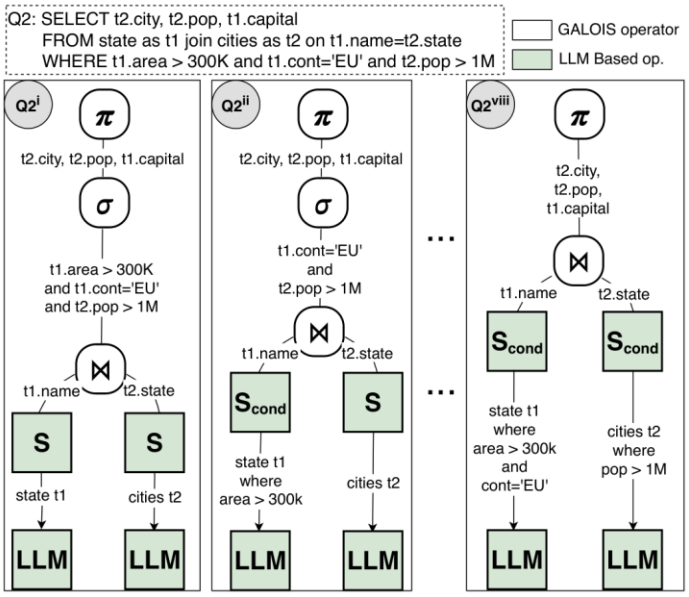


Figure 6 - Different logical plans for the same query Q2.

Starting from the *logical optimizer*, this module dynamically selects the filtering conditions to include in the retrieval phase. Due to the lack of metadata, like indexes or histograms that are often present in traditional RDBMS systems and that are used for the definition of the execution plan, Galois relies entirely on the LLM itself. Thus, they estimate the most efficient logical plan by giving the database schema and the query used to the system, then they ask the LLM to perform a classification task, or rather to return a confidence level, “high” or “low", for each of the atoms present in the WHERE clause. All the atoms with confidence equal to high will be later included in the *Filter-LLMScan*.

For simplicity, they have considered only three possible scenarios:

1. If the model returns no atoms with a “high” confidence label, then Galois will use *LLMScan*, or rather the scan operator with no filtering condition.
2. If it returns a single atom with a “high” confidence label, then Galois will include only that atom in *Filter-LLMScan,* while the other will be applied after the scan execution.
3. If the LLM model returns more than one atom with a “high” confidence label, then all the filtering conditions will be included in *Filter-LLMScan*, regardless of their labels.

Moving forward to the second optimizer, the *physical optimizer* is responsible for selecting the most suitable scan strategy for a given query. As was anticipated earlier, *Key-Scan* uses a sort of Chain-of-Thoughts strategy to enhance the result quality and accuracy; however, in some cases, the additional context introduced by *Table-Scan* can help to increase the overall quality of the retrieved data. To select the right scan methodology, Galois relies once more on the LLM itself by estimating its confidence in retrieving data of attributes involved in the query. To reduce the overestimation by the model, they have implemented a custom function to calculate the overall confidence considering the number of attributes involved in the SQL query, and the formula is the following:

$$conf(q) = LLMconf(keys \mid conds)^n$$

where *n* is the number of attributes involved in the SELECT clause.

Then, if the confidence value is over a certain threshold, tailored specifically to the LLM used, the scan strategy will be *Key-Scan*; otherwise, it will be *Table-Scan*. Usually, the smaller the model is, the higher the threshold value for confidence is.

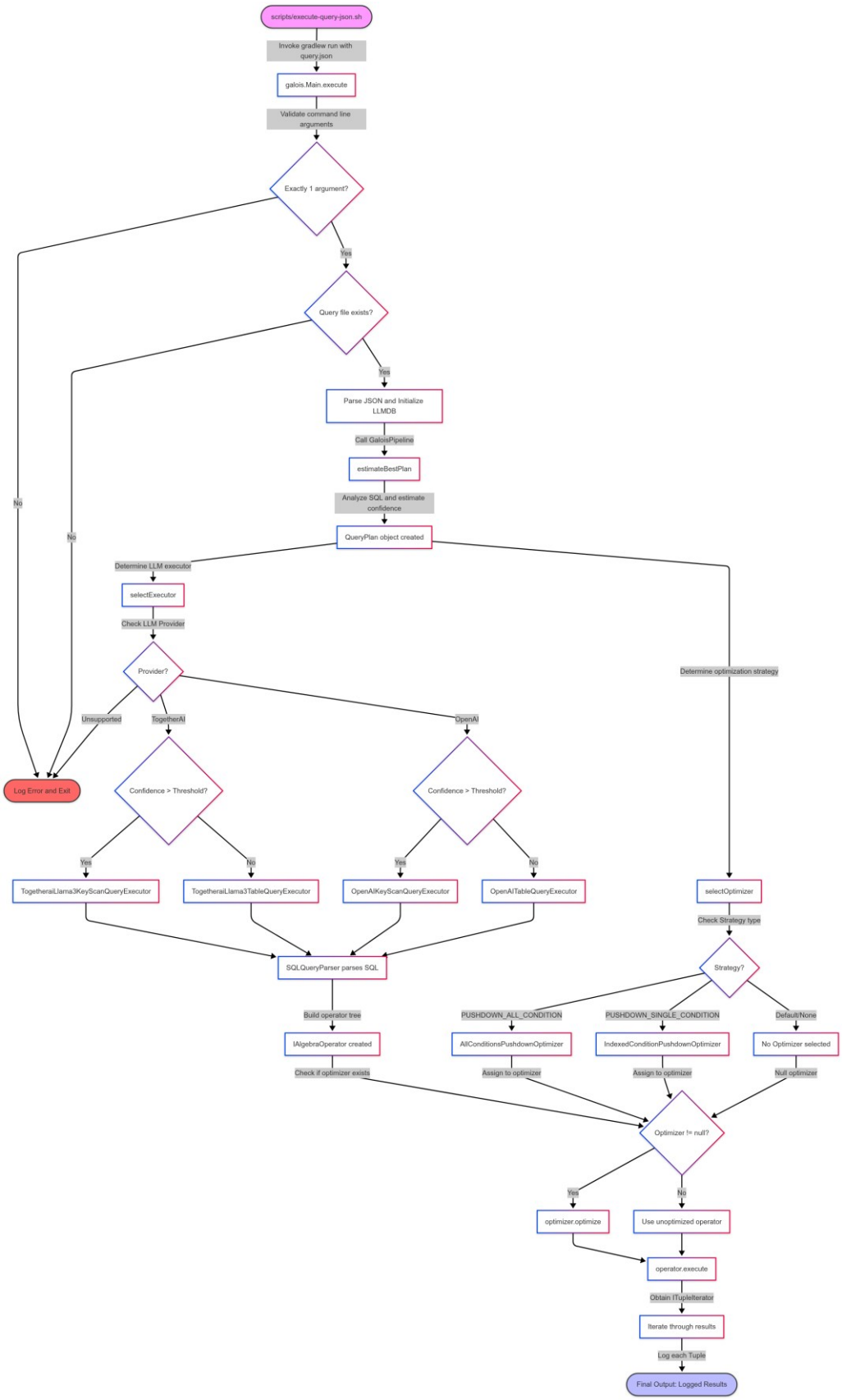


Figure 7 - Galois workflow diagram.

1.4 - Limitations

In conclusion, let us discuss the limitations of Galois. In our opinion, the most important ones are:

- The choice of developing the project in Java instead of in another language that has more tools or functions to operate with LLMs.
- The complexity of the architecture and, indeed, the application itself.
- The lack of comments and Javadoc, which makes it difficult to understand its implementation, leading to complications in extending or expanding this project.
- Manual parsing in case of non-conforming output.
- Static prompt templates for *Table-Scan* and *Key-Scan*.
- Weak management of previously generated content.

Beginning with the language used, Java is one of the most popular programming languages worldwide; however, when it comes to artificial intelligence and machine learning, Python is a better choice because of its extensive libraries and tools made for these fields. For instance, frameworks like *Langchain* or *Llama-Index* are created specifically to manage and work with LLMs, offering ready-to-use designs that make developing AI applications easier. Also, tools like Pydantic are very useful for data validation, furnishing stronger solutions for requesting and validating structured data compared to manual parsing or format constraints, as Galois does.

Secondly, and just as important, is the issue of architectural complexity. Currently, Galois's source code includes over 200 files, excluding test resources and library files. These files are primarily composed of Java code, where interfaces and classes are defined, implemented, and extended. As a result, understanding how each Galois module is designed and what the right sequence of operations is becomes very difficult, especially given the lack of any comments and Javadoc.

Then, another huge limitation of this system is the usage of static prompt templates, in which SQL-like code is injected. Using such a template is convenient but can degrade performance because, although most modern LLMs are trained on SQL syntax and relational databases, they are primarily trained on textual information. Using natural

language prompts can unlock certain portions of the LLMs' internal knowledge that, using only SQL syntax, cannot be accessed.

Last but not least, the management of already-generated content is somehow weak due to the injection of the previous answers in a dedicated section in the user prompt. In longer sessions, smaller models can struggle to identify and retrieve important information, suffering from the so-called *"Lost in the Middle"* phenomenon, in which an LLM has difficulty recovering information located in the middle of long prompts or contexts.

Chapter 2: GaloisPy, the Galois system re-implemented in Python

Although the Galois system implemented a valid and innovative way for retrieving structured data from LLMs' internal knowledge, its implementation is confusing and, therefore, difficult to understand and expand. For these reasons, a re-implementation of the whole system was needed.

In this chapter, we will present GaloisPy, a Python software tool that implements the same functionality provided by Galois but in a cleaner and simpler way, with some modifications or new features designed to improve the general performance.

2.1 - Overview

GaloisPy is the re-implementation of the Galois system in Python; it provides the same functionality as the original, but, in addition, it brings some new features that aim to improve the usability and the overall performance. Its core functionalities rely on some of the most complete and valid Python libraries available at the moment for working and interacting with LLMs: *Langchain* and *Llama-Index*. Furthermore, it uses *Pydantic's BaseModel* classes to produce structured outputs in a more reliable way than asking the model to create a valid JSON as a response to be parsed later.

However, GaloisPy completely re-engineers the overall structure, reducing complexity by refactoring Galois' numerous features by defining a smaller number of modules with a clearer intention of purposes, together with fixing some of the limitations or lack in the original project. Moreover, the whole interaction with such a tool has been simplified, giving access to two main classes, named *LLMRunner* and *RAGRunner*, to be easily used and included in any Python projects.

Therefore, GaloisPy was built following several architectural patterns and design patterns, with the four most relevant being the *Executor/Pipeline Pattern*, the *Repository*

Pattern, *Module-Based Functional Programming*, and *Configuration-Driven Architecture*. A more detailed presentation can be found later in this chapter.

Last but not least, the whole project is completely open source, it's distributed under the *GPL 3.0* license, and the source code within the testing resources is available on a GitHub repository.

2.2 - Architectural patterns and design patterns

As previously anticipated, GaloisPy is based on four main architectural and design patterns; however, a total of nine patterns were applied to this project. Those belong to three distinct categories:

- *Architectural patterns:* They are high-level, proven, and reusable solutions to commonly occurring organizational problems in software architecture. They define the fundamental structural layout, components, and interactions, guiding decisions on scalability, maintainability, and performance.
- *Structural and Creational Patterns:* They belong to one of the three fundamental categories of software design patterns, which focus on organizing classes and objects into larger, more flexible, and efficient structures while simplifying relationships and promoting code reuse.
- *Behavioral patterns:* These patterns focus on improving communication, interaction, and the distribution of responsibilities between objects in a system. Moreover, they define how data flows through the system and how different components collaborate to execute complex tasks.

Specifically, the patterns used during the development of GaloisPy are the following:

- *Module-Based Functional Programming:* It is an architectural paradigm (not an official architectural pattern, but a consolidated architectural archetype widely adopted in the *Functional Programming*[2] context) in which software components are structured into coherent, stateless modules. In this model, logic is encapsulated

[2] It is a declarative paradigm that builds software by composing pure, side-effect-free functions while treating data as immutable.

into pure functions, ensuring a strict separation between behavior and state. For instance, all the functions defined in the *chroma_utils.py* file are standalone with a single, precise responsibility.

- *Configuration-Driven Architecture:* It is an architectural paradigm in which behavior, logical flow, or application structure are not strictly defined in source code, but they change according to external configuration files like JSON or YAML. In fact, GaloisPy utilizes diverse configuration files to modify its behavior or its functionalities. For instance, it accepts JSON files from the user for setting up LLM models.
- *Separation of Concern:* It is a foundational principle in software engineering and involves dividing a software program into distinct sections, each addressing a specific area of interest, also known as a concern or responsibility. In GaloisPy, this is implemented by ensuring that each component belongs to a dedicated domain, such as *ollama_utils*, which manages only the instantiation of Ollama models.
- *Parallel Class Architecture:* In the context of dynamic languages such as Python, the Parallel Class Architecture is characterized by interface symmetry. Although classes such as *LLMRunner* and *RAGRunner* are structurally independent, their functionalities are mirrored. This approach avoids the rigid constraints of classical inheritance, favoring the so-called *Duck Typing*[3], whereby interchangeability is guaranteed by the presence of identical methods.
- *Utility Module Pattern:* It is a structural design pattern that organizes related functions, usually stateless and of general purpose, into a single module. These modules provide useful and reusable tools to other parts of the application, helping to prevent code duplication. In GaloisPy, for example, modules like *chat_utils* offer reusable methods for interacting with the LLM model in different parts of the application's workflow.
- *Dependency Injection:* It is a design principle where an object or function receives its dependencies from an external source rather than creating them on its own.

[3] It is a programming concept, common in dynamic languages like Python, where an object's suitability for a task is determined by its methods and properties (behavior) rather than its actual type or class. Its name comes from the application of the duck test: *"If it walks like a duck and it quacks like a duck, then it must be a duck"*.

Instead of instantiating what it needs, it declares its requirements, and the caller provides them. For instance, in GaloisPy, the *load_llm_config* method of *LLMRunner* asks the user to furnish a valid configuration file to initiate an LLM model.

- *Repository Pattern:* A design pattern that isolates the logic of data access from the business logic of the application. It provides a standardized interface to perform *CRUD* (Create, Read, Update, Delete) or search operations on data, regardless of the underlying storage technology. In GaloisPy, this pattern is particularly relevant in the RAG implementation, where a repository can abstract the complexities of searching through document embeddings, allowing the main logic to request information without knowing the specific details of the vector database or file system.
- *Pipeline Pattern:* An architectural pattern that organizes complex execution processes into small, distinct phases or steps. Each step receives input from the previous one, processes or modifies it, and then propagates the result to the next phase. The entire process is linear and unidirectional, facilitating reuse and testing. For instance, the whole retrieval process of GaloisPy is decomposed into several steps, each of which involves one or more data manipulation steps.
- *Template Method Pattern:* It is a behavioral pattern used to define the fixed skeleton of an algorithm while allowing specific steps to be tailored by external inputs. In GaloisPy, this pattern ensures that the execution flow, like in the RAG execution sequence, remains consistent, while the specific behavior of each phase is dynamically determined by the *config.ini* files. This allows the application to maintain a rigid, reliable process architecture while still being highly adaptable through configuration.

2.3 - Architecture

2.3.1 - Base structure

The GaloisPy's design can be divided into three major modules, following the structure of the repository:

- *galoispy\base:* It contains all the files related to the execution of the standard mode, i.e., the execution mode in which data is retrieved from the LLM's internal knowledge. Inside, there is the Python class that implements the whole logic, named *LLMRunner*, and some utility functions tailored to this modality. Those files are contained inside a sub-folder named *utils*.
- *galoispy\rag:* Inside, there are the files for the execution of the RAG mode, which, unlike the standard mode, knowledge comes from a set of documents that the user furnishes to the system, rather than from the LLM itself. As in the standard mode, there is a class that encapsulates the whole logic named *RAGRunner*, and all the helper functions and modules are contained in the *utils* sub-folder.
- *galoispy\utils:* All common modules for both modalities are contained in this folder. Inside, there are the files that implement common functionalities, such as the definition of the execution plan starting from the SQL query, the configuration, and the instantiation of an LLM model, or the management of the retrieval process. All of them present a list of functions that implement a specific functionality, as defined by the *Separation of Concern* and by the *Module-Based Functional Programming* pattern.

Additionally, two extra folders provide important utilities for the program, which are:

- *galoispy\exception:* It contains the implementation of two custom exceptions, which are: *NotStructuredAnswerError* and *OpenAIQueryError*. The first one is thrown when the LLM does not provide a structured output, while the second is used when an OpenAI model raises an exception, often related to filtering policies or to usage limitations.
- *galoispy\models:* Inside, there are the JSON schemas used during the logical and physical optimizations or during the prompt rewriting. These files report the structure of Pydantic's BaseModels that are used to ensure that the LLM model produces the expected structured output.

README.md

```
# Repository structure
```
GaloisPy/
├── .gitignore
├── README.md
├── pyproject.toml (project configuration file)
├── LICENSE (license file)
├── src/
│   ├── galoispy/
│   │   ├── base/
│   │   │   ├── config.ini
│   │   │   ├── llm_runner.py (the LLM runner class)
│   │   │   └── utils/
│   │   │   └── ... (galoispy utility files for this mode)
│   │   ├── rag/
│   │   │   ├── config.ini
│   │   │   ├── rag_runner.py (the LLM runner class)
│   │   │   └── utils/
│   │   │   └── ... (galoispy utility files for this mode)
│   │   ├── exception/
│   │   │   └── ... (galoispy custom exception)
│   │   ├── models/
│   │   │   └── ... (pydantic schema files)
│   │   ├── utils/
│   │   │   └── ... (galoispy llm utility files)
│   │   ├── __init__.py
│   │   └── config.ini
```
```

Figure 8 - GaloisPy repository structure.

2.3.2 - Components interaction and workflow

Once we have explained the general architecture, let us now describe how GaloisPy works and what its general workflow is from the input SQL query to the final output.

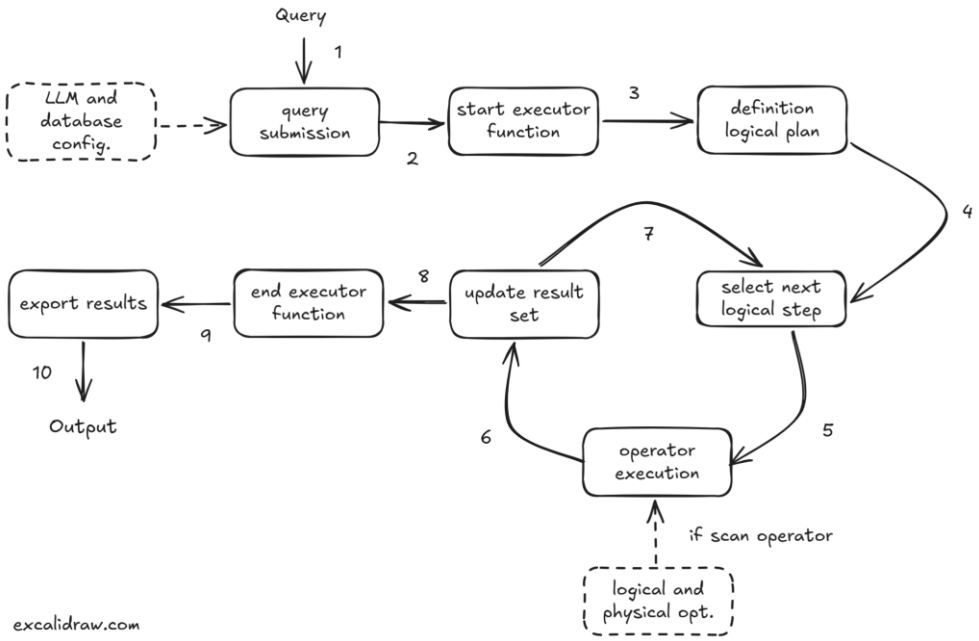


Figure 9 - GaloisPy workflow from the query submission to the final output.

Once the user has correctly instantiated and configured one of the two runner classes, they can submit queries to the module for execution by passing the SQL as an argument to the runner's function *query_runner*. Then, GaloisPy invokes the *executor* function for executing the query, passing the query, the database schema previously loaded, and some of the LLM model's parameters, like the tokenizer used or the size of the context window.

After that, the executor function loads the setup from the two config.ini files, including details such as whether to use an enhancement, the prompts for the scanning and optimization phases, and more. Then it defines the logical plan by calling the function *def_logic_plan* and receiving a list of *LStep* objects as output, where each step represents an operation in the logical pipeline. Next, it begins executing the logical plan by performing one step at a time.

After the logical pipeline completes, it returns to the caller the result set as a list of *dict* objects, the number of tokens used, and the number of iterations required during the scan operation. Finally, GaloisPy calculates the execution time, both the CPU time and the execution time, exports the result set in CSV format, and returns to the caller the result set. Moreover, if the user is interested, they can obtain execution metrics from GaloisPy, which are the number of tokens used, the number of iterations performed during the scan operations, the CPU time, and the execution time.

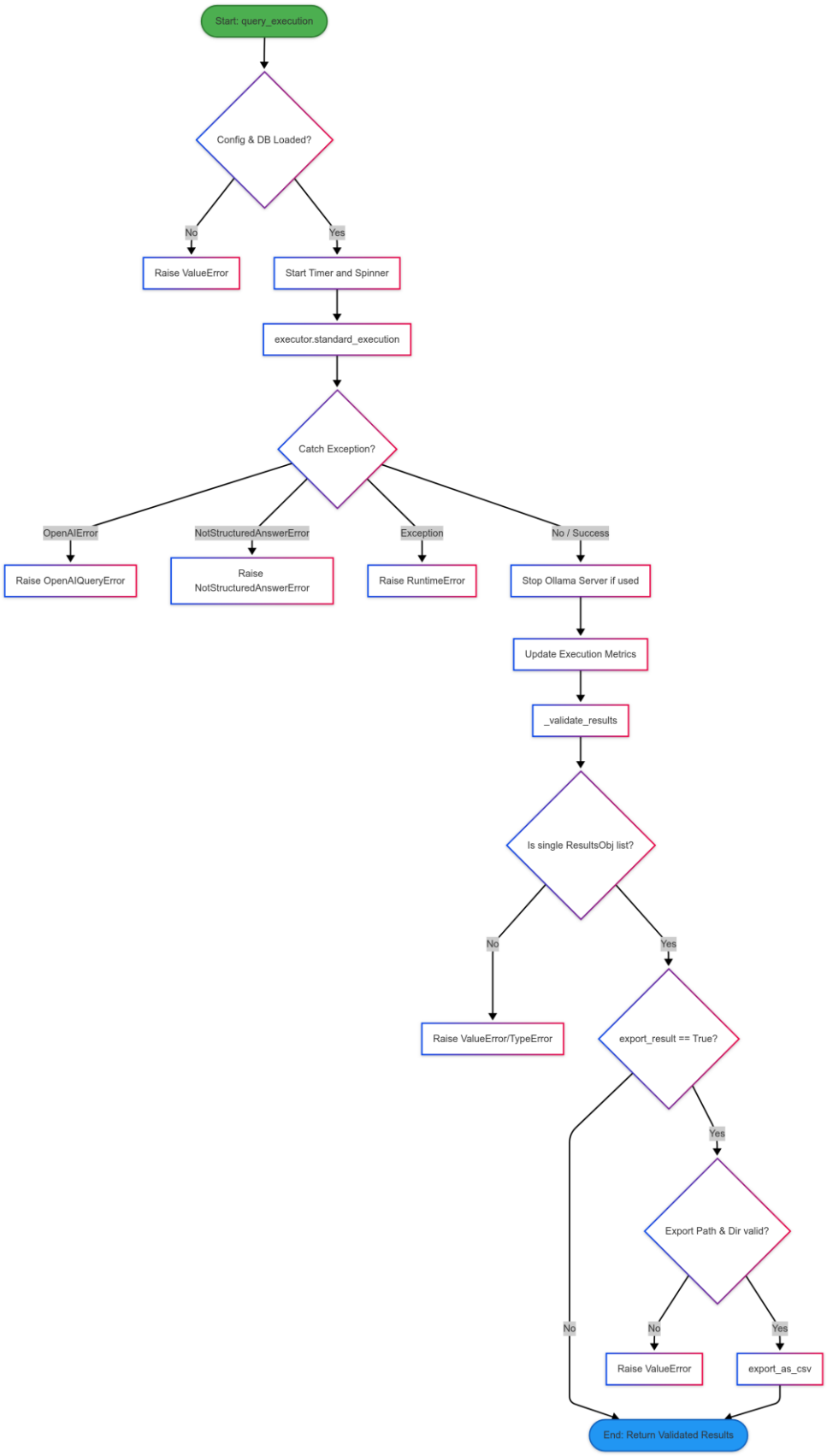


Figure 10 - Query execution workflow.

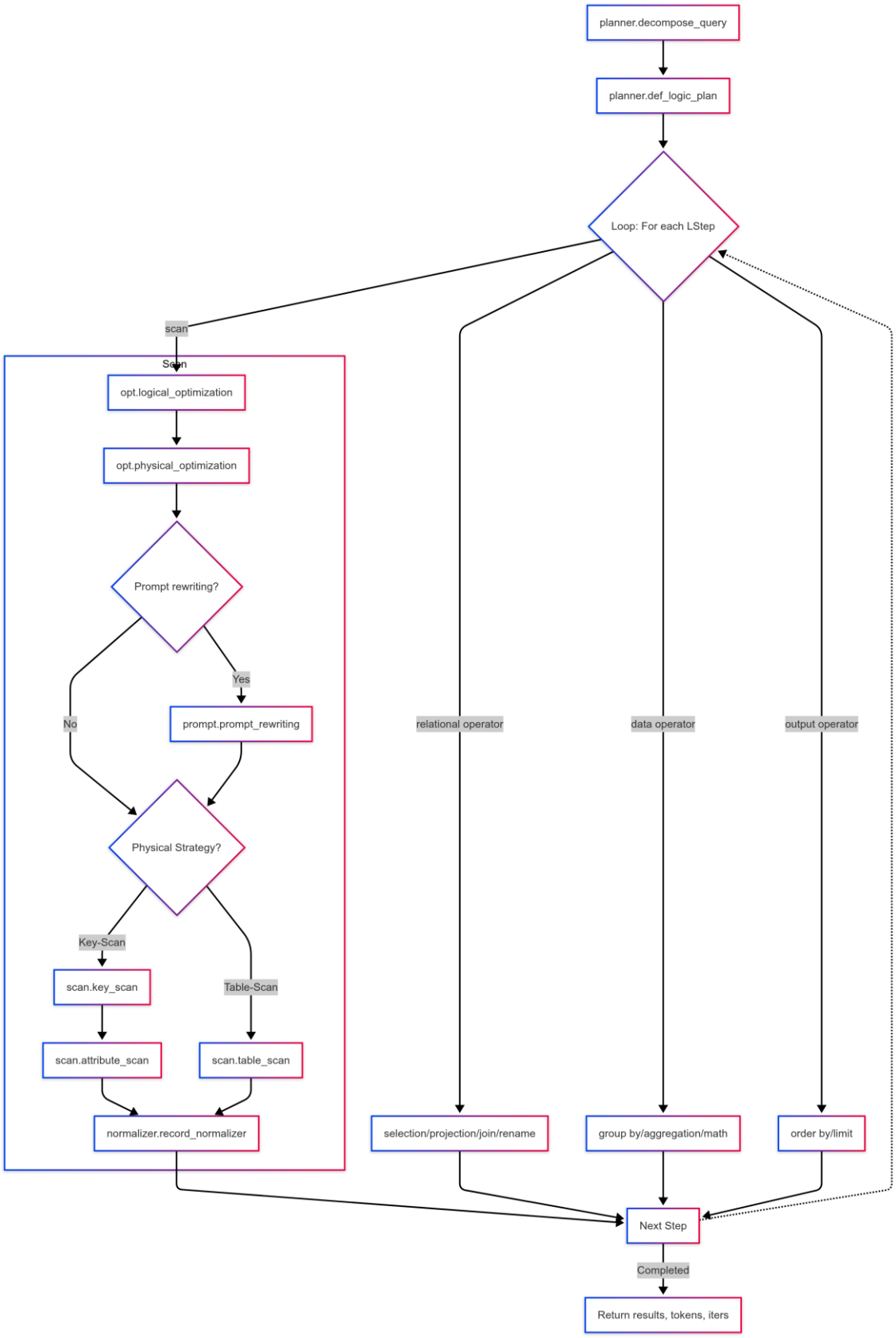


Figure 11 - Logical plan execution.

2.4 - Optimizations

The GaloisPy system not only reimplements the same functionalities of Galois, but it also includes some modifications and optimizations designed to improve performance over the original system, and their effectiveness will be later presented within the test results.

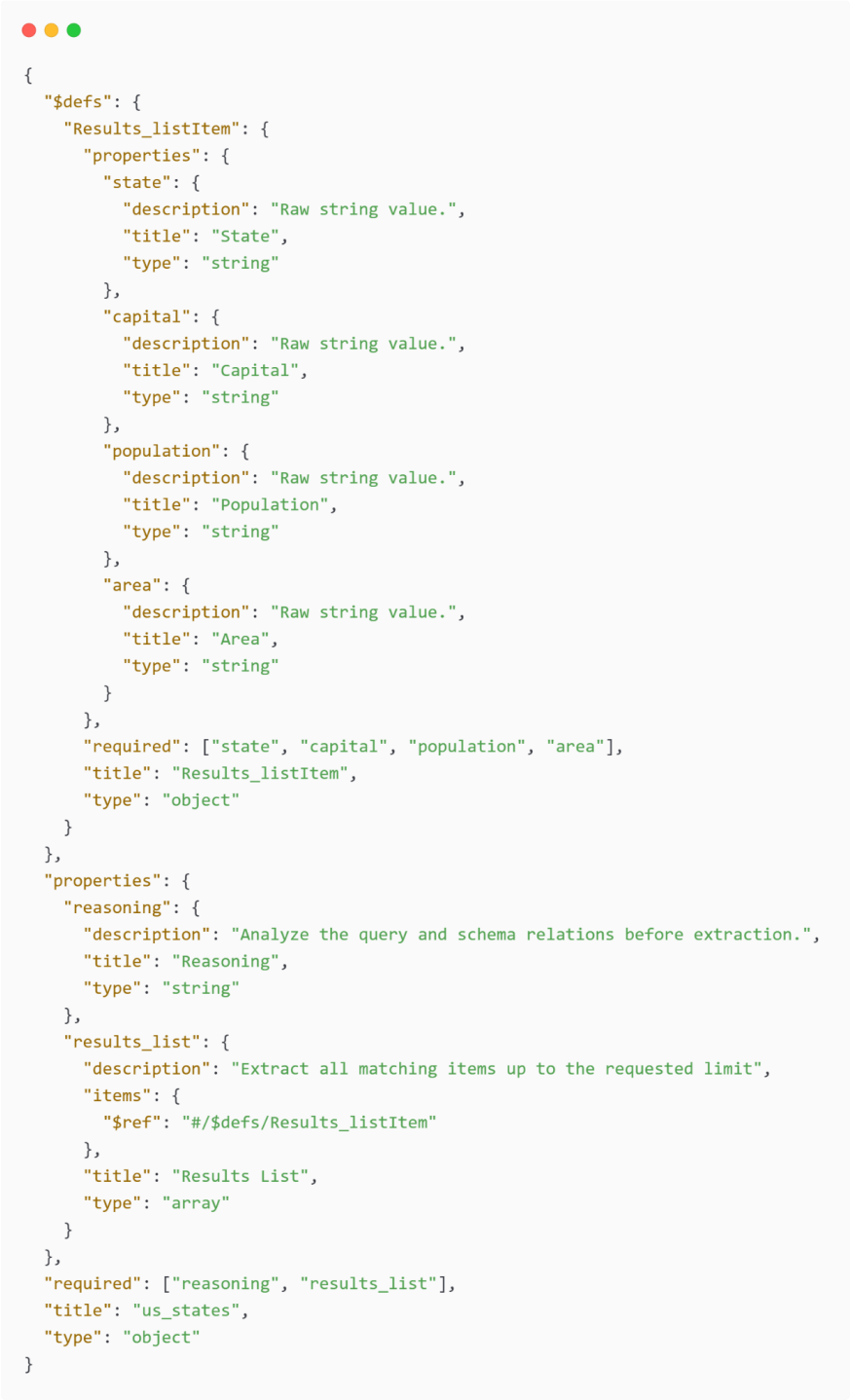
In this paragraph, we will cover solely the most important one, and those modifications are:

- The implementation of a real chat history using *Llama-Index* and *Langchain* libraries.
- The integration of *Pydantic's BaseModel* classes, which are defined automatically based on the input query, without the need for users to define them manually. These classes guarantee that the LLM model always returns the expected structured output.
- The introduction of an extra section in the *BaseModel* that allows the LLM to reason before generating the output.
- The support for the reranking models in the RAG pipeline.
- The implementation of some enhancements designed to improve the quality of the retrieved data. These are: a retry mechanism in case of empty response, additional description fields in the database schema to furnish a clearer context, and a prompt rewriting method from SQL-like queries into natural language questions.

Starting with the chat history, this was introduced to improve the management of the already generated responses. By defining a chat history, the large language model can understand and keep track of the progression during the retrieval process, reducing the number of already generated tuples and, as a consequence, improving the efficiency of the process by reducing the number of iterations needed. Moreover, a system prompt is included to deliver a comprehensible role and cleaner instructions about how to generate the desired outputs. On top of that, it also manages memory initialization, memory refactoring to avoid context window overflow, and removal from the chat history of the thinking sections of models that support this kind of feature, like the Qwen3 family models, to reduce the number of tokens stored in it. With this enhancement, we aim to deliver to the model a clearer and more structured record of the whole conversation,

reducing the so-called *"Lost in the Middle"* phenomenon, allowing it to better understand what it has already generated and what task to perform.

The second major modification ensures that the LLM model returns structured outputs. To accomplish this, GaloisPy uses *Pydantic's BaseModel*, which allows users to define custom classes with different attributes and types. Thanks to solid validation during the instantiation process, it guarantees that newly created objects conform to the *BaseModel's* schema. Although this type of class is powerful, users must first define a schema. Since a valid schema is required for each query, asking the user to provide one is impractical; therefore, automation is necessary. To overcome this limitation, we created a module named *dynamic_pydantic* that defines a *BaseModel* class containing the attributes listed in the SELECT clause together with their type, retrieved from the database model, starting from the SQL query, which can be the input query or an intermediate query generated by GaloisPy itself, and the database JSON schema. With this module, GaloisPy can leverage *Pydantic's* capabilities without requiring additional user input. Additionally, *Langchain* seamlessly integrates with *Pydantic*, providing the LLM with the desired *BaseModel* schema and handling the entire validation process to ensure the output conforms to the input schema.



```
{
  "$defs": {
    "Results_listItem": {
      "properties": {
        "state": {
          "description": "Raw string value.",
          "title": "State",
          "type": "string"
        },
        "capital": {
          "description": "Raw string value.",
          "title": "Capital",
          "type": "string"
        },
        "population": {
          "description": "Raw string value.",
          "title": "Population",
          "type": "string"
        },
        "area": {
          "description": "Raw string value.",
          "title": "Area",
          "type": "string"
        }
      },
      "required": ["state", "capital", "population", "area"],
      "title": "Results_listItem",
      "type": "object"
    }
  },
  "properties": {
    "reasoning": {
      "description": "Analyze the query and schema relations before extraction.",
      "title": "Reasoning",
      "type": "string"
    },
    "results_list": {
      "description": "Extract all matching items up to the requested limit",
      "items": {
        "$ref": "#/$defs/Results_listItem"
      },
      "title": "Results List",
      "type": "array"
    }
  },
  "required": ["reasoning", "results_list"],
  "title": "us_states",
  "type": "object"
}
```

Figure 12 - Example of Base Model schema in JSON format about US states.

The third major improvement is the addition of an extra section, named reasoning, to the BaseModel schema. This allows the model to think about the request before generating the answer. This small yet important modification enables the LLM model to analyze the request carefully before answering, resulting in fewer hallucinations and more accurate

responses. Since LLMs generate text one token at a time, placing the reasoning field first causes the model to consume tokens more closely related to the actual answer. Due to its autoregressive nature, the model shifts the probability distribution towards more relevant tokens. This technique is especially effective for smaller models, which benefit greatly from this extra step, improving their accuracy.

dynamic_pydantic.py

```
# ...
fields["reasoning"] = (
    str,
    Field(..., description="Analyze the query and schema relations before extraction.")
)
# ...
```

Figure 13 - The reasoning field.

Moving forward, another improvement is the support for reranking models, at the moment limited to only the ones available on Hugging Face, in the RAG pipeline. This simple but important improvement aims to improve the quality of the documents retrieved from the vector database, providing the LLM model with more relevant documents and, consequently, improving the quality of the results generated.

Last on the list to improve the retrieval capabilities of the system, we have introduced three main improvements: a retry mechanism in case of an empty response, additional description fields within the database definition, and an LLM-based prompt rewriting tool to translate SQL-like requests into natural language questions. All these tools can be used simultaneously, and the user can decide which of these to use by modifying the configuration file of the *Runner* class used. Starting with the retry method, it can happen occasionally that, when the LLM model is not sure about the response to provide, it prefers not to return any result. Giving it a second chance to generate the response again can improve the overall recall, obtaining a more complete result set. Moving to the second improvement, the introduction of description fields embedded in the database schema aims to increase the understanding of the context by furnishing the model with a complete explanation about the meaning and the purpose of every attribute and table defined in the schema. Information reported within descriptions could be, for instance, explanations related to enumerated attributes, such as the gender, or guidance about specific protocols,

such as the IATA code[4] or the ICAO code[5]. Finishing with the prompt rewriting tool that translates the SQL syntax into natural language questions. Although modern large language models are trained on SQL materials, they, however, tend to perform better with natural language. For this reason, we developed a function named *dynamic_prompt* that rephrases the prompts used by *Table-Scan* and *Key-Scan*, both of which use SQL-like syntax, into natural language questions while maintaining the original prompts' exact meaning.

2.5 - Distribution and usage

GaloisPy is distributed on GitHub and can be ported and installed easily on any machine thanks to a *wheel* package release. Once installed, it can be imported and used as any other library installed through the Python package manager.

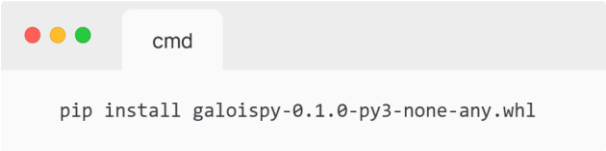


Figure 14 - Process to install the package locally.

GaloisPy has two modes: base and RAG. The base mode uses the LLM's internal knowledge, while the RAG mode relies on documents provided by the user. The configuration process is straightforward. Once one of the two classes is instantiated, a method allows the user to load the LLM's configuration; at the moment, it supports only Ollama models and OpenAI models through Microsoft Azure, with all its parameters, such as the tokenizer model, the model temperature, and the top-p value; while another one allows the user to furnish the database schema.

After correctly loading these two configurations, the user can provide the model with an SQL query and receive a list of dictionaries containing the query results as output.

[4] It is a unique three-letter geocode designating many airports, cities (with one or more airports) and metropolitan areas (cities with more than one airport) around the world, defined by the International Air Transport Association (IATA).

[5] It is a four-letter code defined by the International Civil Aviation Organization (ICAO) used by air traffic control to identify airports or other aeronautical facilities such as weather stations.

Additionally, the results can be exported as CSV files for broader integration with existing applications.

main.py

```python
from galoispy import LLMRunner

# Create an instance of the desired runner class
runner = LLMRunner()
# Load the LLM configuration and the database schema
runner.load_llm_config(config_schema=llm_configuration_file, env_path=env_path)
runner.load_database(db_schema=db_configuration_file)
# Run a query
results = runner.query_execution(query=query, export_result=True, export_path=folder_path, filename=filename)
# Get performance metrics
metrics_data = runner.get_execution_metrics
```

Figure 15 - How to import and use GaloisPy.

Chapter 3: Internal knowledge experiments

To objectively assess the validity of our re-implementation, we have evaluated GaloisPy using the original Galois test suite. In this first round of experiments, we have tested GaloisPy capabilities based only on LLM's internal knowledge (IK). Then, we compared the performance of both implementations, highlighting each one's strengths and weaknesses.

In this chapter, we will present the experimental setup, covering the datasets used and their content. We then present the results obtained with GaloisPy, followed by a comparative analysis of both systems, focusing on the specific improvements, potential trade-offs, and the overall effectiveness of the optimization methods introduced in our version.

3.1 - Experimental setup

For these experiments, we have taken the same datasets used by the Galois team in their paper titled *"Logical and Physical Optimizations for SQL Query Execution over Large Language Models"*. These datasets are:

- *Flight:* This dataset contains information on flights, including related data such as airports and airline companies. The dataset belongs to *Spider*, a large-scale, complex, and cross-domain semantic parsing and text-to-SQL dataset. This dataset is then divided into two sub-datasets: *Flight-2* and *Flight-4*. The first one is tailored to the USA, including airports located in its territory, flights that take off or land at USA airports, and USA airline companies; meanwhile, the second one contains data from around the world.
- *Geo:* Another dataset from Spider that contains information about the United States of America's geography, including cities, states, lakes, rivers, mountains, and many more.

- *Movies:* It contains data on movies extracted from the IMDb database, one of the most popular and authoritative sources of film information. It reports data such as movie director, production year, duration, and more.
- *Presidents:* A web-scraped dataset from Wikipedia about government presidents of Venezuela and the United States of America. This dataset reports information such as the president's full name, the country, the start and end years of the presidential mandate, and more.
- *World:* Another dataset of Spider regarding demographic, geographic, and political data on different countries.

Dataset name	Dataset source	# of queries	Avg. expected cells	Experiment type
Flight	Spider	6	267.5	IK
Geo	Spider	32	22.8	IK
World	Spider	4	33.2	IK
Movies	IMDB	9	54.7	IK
Presidents	Wikipedia	26	42.2	IK

Table 1 - Statistics about IK datasets.

Regarding the LLM model, we have chosen to use a more budget-friendly and much less resource-intensive model than *Llama 3.1 70b*, *GPT-4.1 nano*, the entry-level model of the OpenAI family.

Model	GPT-4.1 nano
Temperature	0.0
Top-p	0.01
Frequency penalty	0.0
Presence penalty	0.0
Max token	13107
Content window size	1000000
Tokenizer (*tiktoken*)	cl100k_base
Physical opt threshold	0.8

Table 2 - Model configuration.

To evaluate the effectiveness of the GaloisPy system compared to straightforward, more direct approaches, we used the same two baselines of the Galois test suite, but adapted them to our new architecture. These are:

- *Natural Language (suffix NL):* It combines the strengths of Pydantic's structured output and natural language prompts. It is used to evaluate the effectiveness of natural language prompts.

- *SQL:* It queries the LLM using only SQL statements, and, like the NL mode, it relies on Pydantic for structured answers. It is used to evaluate the efficacy of SQL prompts.

Moreover, we evaluated our system, GaloisPy, using the same four configurations tested in the original experiments, which are the following:

- *Without Optimization (suffix WO):* It does not perform the logical or the physical optimization, and it uses the *Key-Scan* strategy to query the LLM.
- *All Condition Pushdown (suffix AP):* It uses the *Table-Scan* strategy, plus it pushes into the scan operator all the filtering conditions, or thus all the WHERE clause's conditions.
- *Selective Condition Pushdown (suffix SP):* It applies only the logical optimization to select the best filtering conditions to include in the prompt during the retrieval phase. It uses the *Table-Scan* strategy.
- *Full Optimization (suffix FO):* It performs both the logical optimization and the physical optimization, aiming to improve the general performance. It represents the true GaloisPy model.

Lastly, we have tested GaloisPy with all its enhancements: the addition of description fields, the ability to retrieve data after the model returns an empty response or only duplicates, and prompts rewriting. We tested all possible combinations to determine which led to increased performance and which did not. Furthermore, we studied how these improvements interact with each other and which GaloisPy modes benefit more from a particular enhancement than others.

3.2 - Evaluation metrics

To evaluate the performance of the system, we kept the same metrics used by the Galois team in the original paper, and these are:

- *F1 Cell:* It is the harmonic mean of precision and recall across the set of cells in the result set relative to the set of cells in the ground truth. This metric evaluates the quality of the result set at the cell level.

$$F1\ Cell = \frac{2 \times precision(result\ set, ground\ truth) \times recall(result\ set, ground\ truth)}{precision(result\ set, ground\ truth) + recall(result\ set, ground\ truth)}$$

- *Tuple Cardinality:* It is the ratio of the number of tuples in the result set to the number of tuples in the ground truth set, assessing the system's capabilities of producing the right number of tuples.

$$Tuple\ cardinality = \frac{min(size_{result\ set}, size_{ground\ truth})}{max(size_{result\ set}, size_{ground\ truth})}$$

- *Tuple Constraint*: It measures the fraction of the tuples in the result set that are present in the ground truth, at the tuple level. This metric, which is stricter than *F1-Cell*, evaluates whether tuples have the same schema, cardinality, and cell values.

$$Tuple\ Constraint = \frac{size(result\ set \cap ground\ truth)}{size(ground\ truth)}$$

- *Avg Score:* The average of the three previous metrics used to measure the overall quality of the result set.

$$Avg\ Score = \frac{F1\ Cell + Tuple\ Cardinality + Tuple\ Constraint}{3}$$

Since both the F1-Cell and the Tuple Constraint rely on exact equality comparisons and LLMs can produce similar results, they relaxed this constraint, introducing a similarity mechanism. They set a maximum similarity threshold of 10% with respect to the expected value. For strings, they used the edit distance, also known as the Levenshtein distance; while for numerical values, they used the absolute difference between the values. This feature enables users to accurately compare similar values that have the same meaning. One notable example is distinguishing between "Mark Zuckerberg" and "Mark E. Zuckerberg," the chief executive officer of Meta.

In addition, we have analyzed execution time, token usage, and the number of queries correctly executed to truly evaluate the capabilities of both systems.

To test the GaloisPy system, we have developed a Python script, named *test_runner.py*, and available on the test branch on GitHub, that automatically executes all tests with a specified configuration, customizable through in-line arguments.

3.3 - Results and comparisons

Since Galois was not tested on *GPT-4.1 nano*, no data on its performance with that model is available; therefore, no meaningful comparison can be made between the two. To address this issue, we ran the Galois test suite, but, unfortunately, we have found that the performance metrics reported by their system were inaccurate, especially the ones that used the Edit distance to compute similarity scores. In fact, regardless of the LLM response, we found that all similarity scores were most of the time 1, rather than the correct value.

```
--------------------------------------------------------------------------------
Flight_2-Q1 Experiment-nl
Expected results:
[[oid:0, call_sign:jetblue]]
Actual results:
[[oid:1, message:please provide the details or question you'd like me to respond to in json format.], [oid:5, message:please provide the specific information or
question you'd like me to respond to in json format.], [oid:2, response:please provide the specific information or question you'd like me to respond to in json
format.], [oid:4, status:error, message:no content provided to generate a json response.], [oid:6, status:error, message:no specific content provided to generate
a json response.], [oid:3, status:error, message:no specific data or question provided to generate a json response.]]
Scores:
Query Executor: OpenAINLQueryExecutor@430c0a3d
SQL Query: SELECT call_sign FROM target.usa_airline_companies WHERE airline='jetblue airways'
CellPrecision: 0.0
CellSimilarityPrecision: 0.16666666666666666
CellRecall: 0.0
CellSimilarityRecall: 1.0
F1ScoreMetric: null
CellSimilarityF1Score: 0.2857142857142857
TupleCardinality: 0.16666666666666666
TupleConstraint: 0.0
TupleSimilarityConstraint: 1.0
LLM Total Requests: 10
LLM Total Input Tokens: 276.0
LLM Total Output Tokens: 460.0
LLM Total Tokens: 736.0
LLM Time (ms): 2745
```

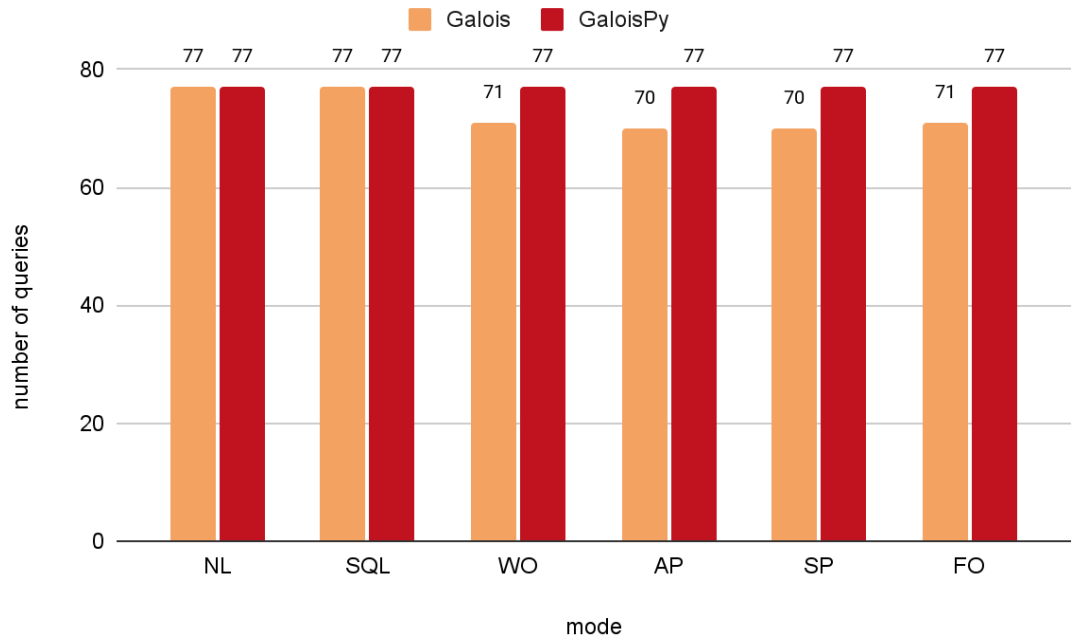
Figure 16 - Example of wrong metric score calculation. In this picture, it is possible to notice how "CellSimilarityRecall" and "TupleSimilarityConstraint" metrics are evaluated to 1 (100%), even if the LLM's output is a string error message.

Since their metrics were inaccurate, we recalculated them from the generated output files. First, we manually retrieved all the correct CSV files from the output folder, which contains hundreds or thousands of files. Then, we cataloged them in the appropriate Galois mode, such as WO, SP, etc. Then, we extracted all information on token usage and execution time of the previously selected models' runs from the XLSX files. Finally, we developed a program that recalculated all metrics from CSV output files.

All Galois and GaloisPy results tables are available in a Google Drive folder[6].

3.3.1 - Galois vs GaloisPy

Let us now compare the performance of our re-implementation, GaloisPy, against the original system, Galois, over the different metrics previously listed. Before we begin, it is important to note that the following metrics are calculated over the set of actual executed queries, and this number might differ from the expected one, 77. Below are reported the number of queries executed per mode for both Galois and GaloisPy.

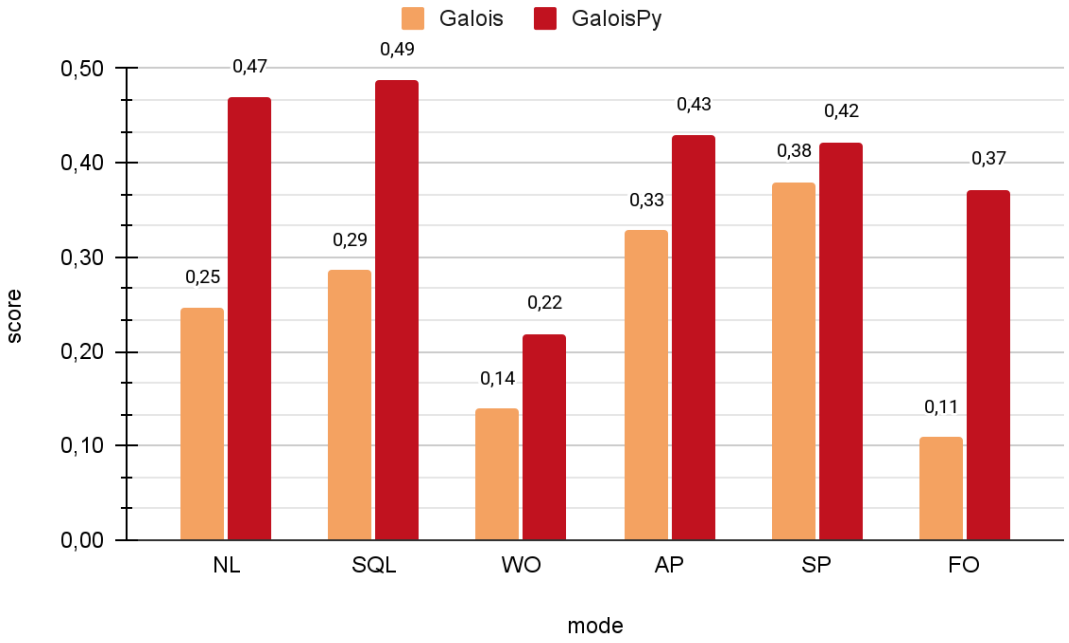


Graph 1 - Number of queries executed. Expected 77.

Starting with the F1 Cell metric, we can see that all GaloisPy modes tested have performed significantly better than their Galois counterparts, especially FO, SQL, and NL. This indicates that our improvements, such as the additional reasoning fields or the enhanced system prompt, have helped GaloisPy to achieve higher scores. Additionally, the FO mode has performed slightly worse than other GaloisPy modes, like AP and SP, suggesting that the Table-Scan approach may yield slightly better results. Nevertheless,

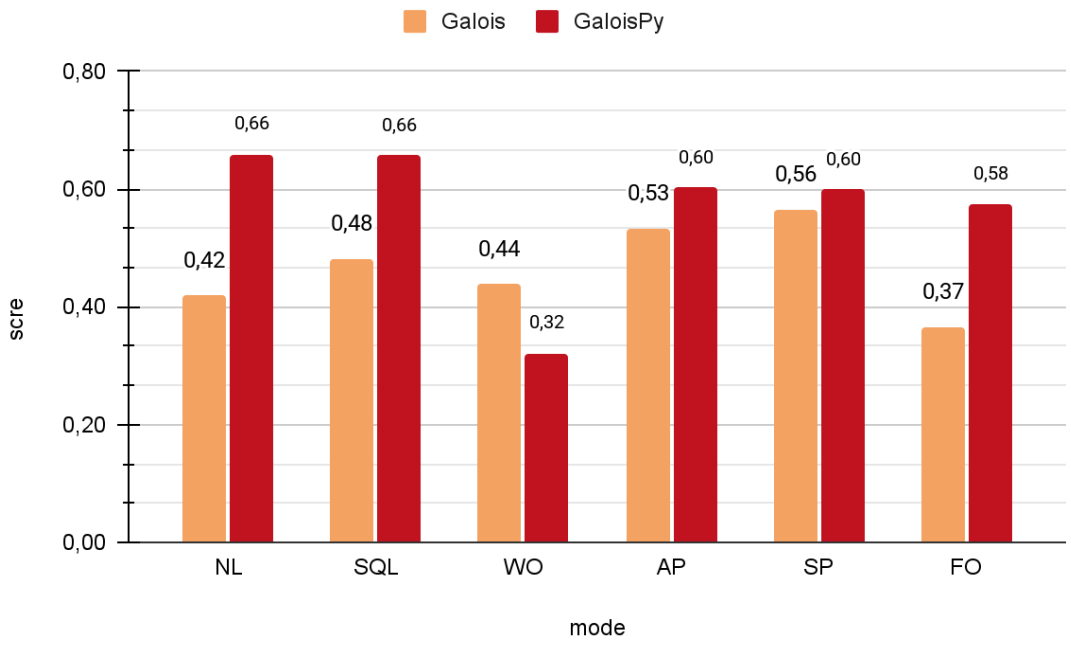
[6] https://drive.google.com/drive/folders/18Vbi7cbcE2QZnbcoVtUFzWiiWEB2Pf9m?usp=sharing – the link may change, please check the *README.md* file on the test branch on GitHub.

more direct approaches, such as SQL and NL modes, have shown superior performance compared to the more sophisticated GaloisPy modes.



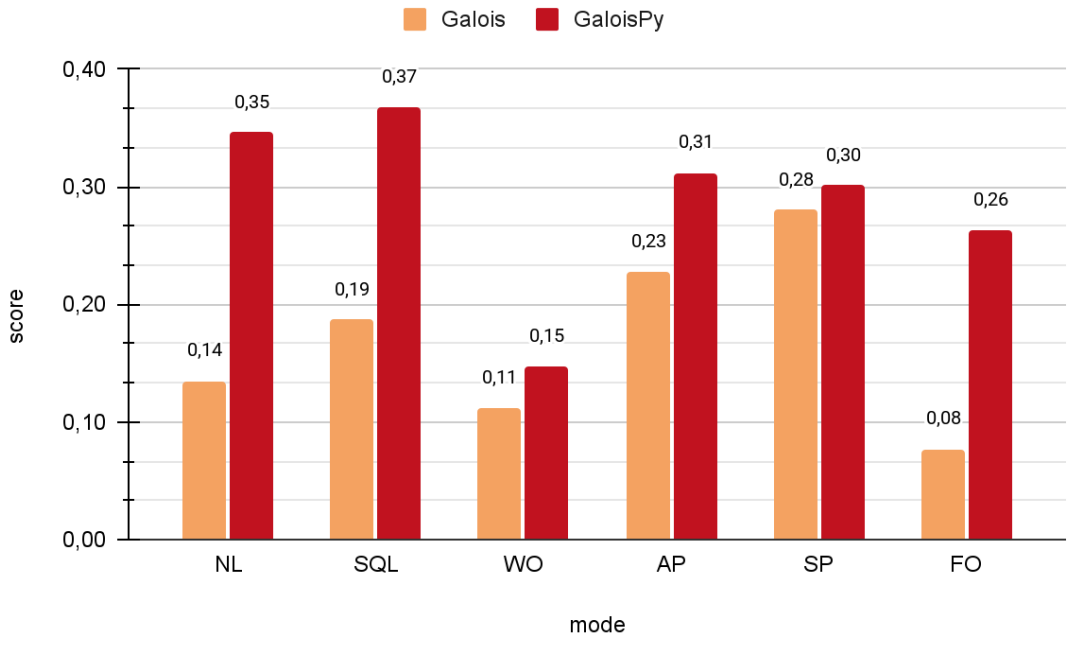
Graph 2 - F1 Cell metric scores of Galois and GaloisPy over different modes.

Moving on, we can notice that the *Tuple Cardinality* metric has also increased, especially for the FO, NL, and SQL modes. Interestingly, almost any GaloisPy modality has achieved at least 60% without any optimization, demonstrating strong retrieval capabilities for such a small model as *GPT-4.1 nano*. Additionally, both AP and SP have achieved the same score, suggesting that including all filtering conditions or only the most confident ones during the scan phase does not affect the overall cardinality significantly. Regarding the FO mode, it has achieved slightly worse results compared to AP and SP. For this metric, however, the NL and SQL modes performed better than all the other tested GaloisPy modalities.



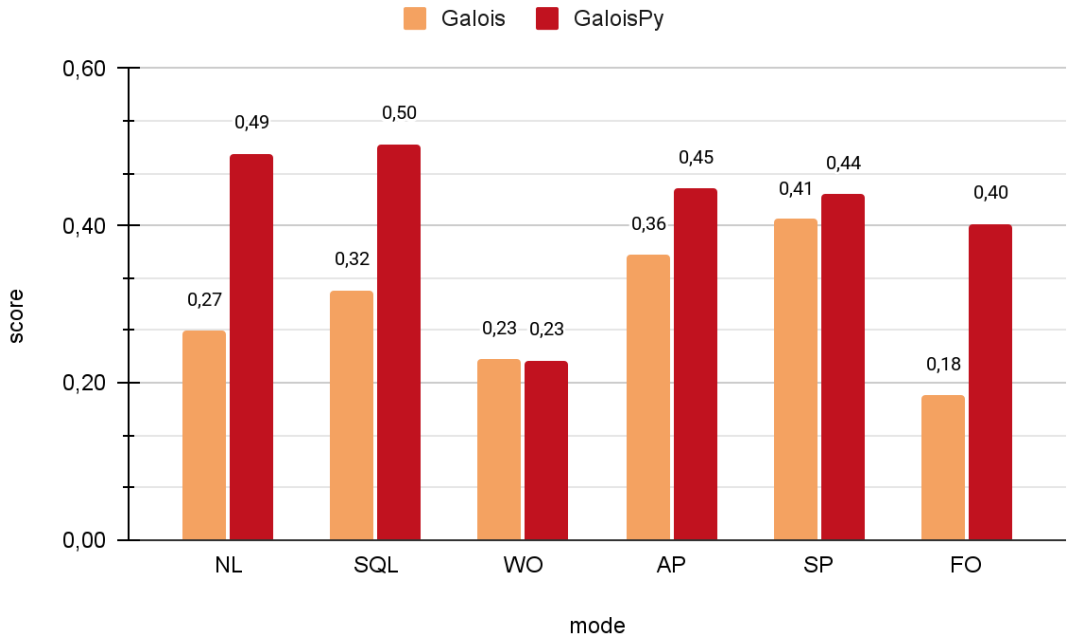
Graph 3 - Tuple Cardinality metric scores of Galois and GaloisPy over different modes.

Proceeding to the next metric, *Tuple Constraint*, we can see that the overall quality of the retrieved tuples has improved across all GaloisPy modes, with AP performing slightly better than SP. Nevertheless, FO still shows negligibly worse performance, which once again highlights that highlighting the Table-Scan strategy can lead to higher performance.



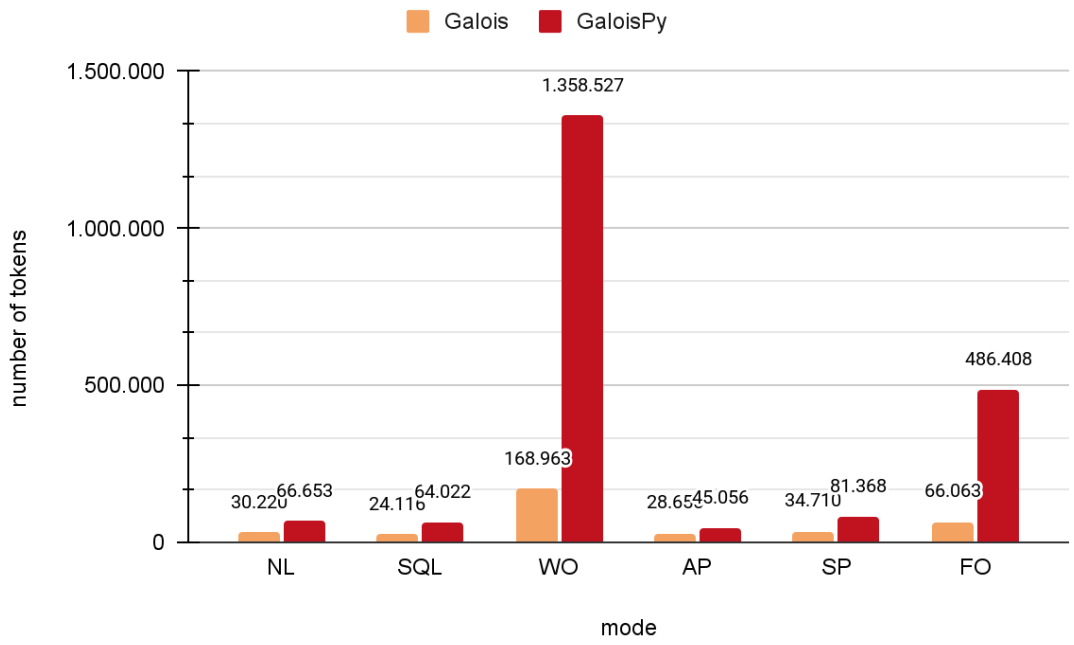
Graph 4 - Tuple Constraint metric scores of Galois and GaloisPy over different modes.

Continuing with the next metric, Avg Score, we can see that GaloisPy has significantly outperformed Galois in the NL, SQL, and FO modes, while it has a higher, albeit similar, performance in the AP, SP, and WO modes. These results demonstrate the superiority of our re-implementation of the Galois system in terms of result quality. This is likely due to the integration of a clearer, more structured memory chat, a more accurate system prompt, and structured outputs through Pydantic's BaseModels. These improvements have helped the model generate more accurate and complete results.



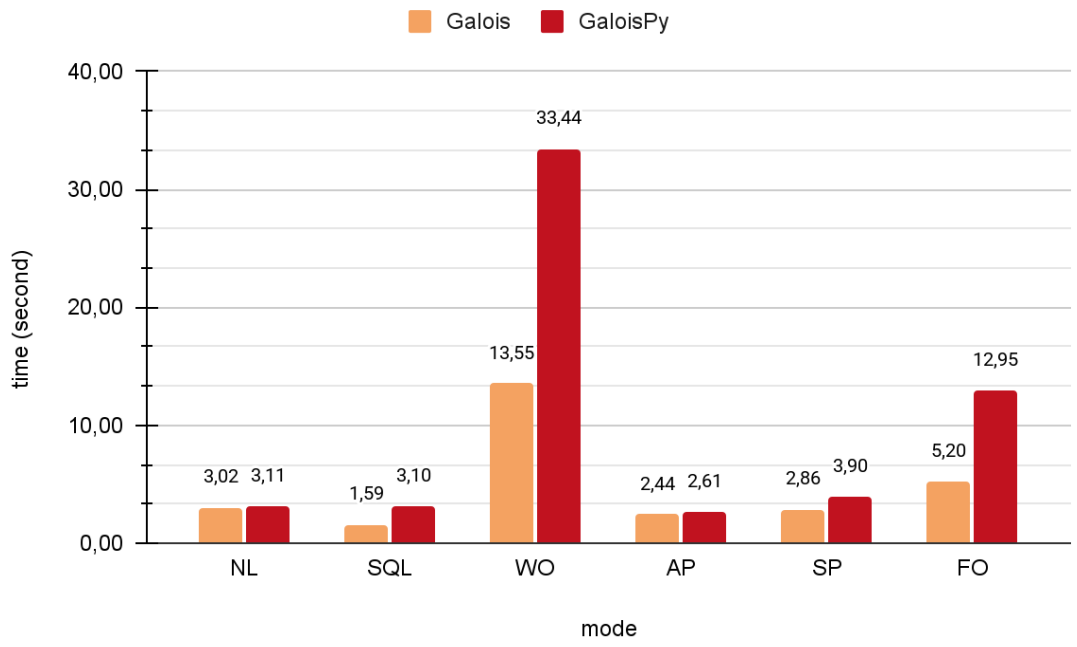
Graph 5 - Average Score metric scores of Galois and GaloisPy over different modes.

Regarding our third-to-last metric, token usage, GaloisPy utilized a significantly higher number of tokens than Galois, particularly in FO and WO modes, both of which exceeded one million tokens. This excess can be partially justified by GaloisPy's implementation of more sophisticated chat memory management. This method requires more tokens, but it guarantees a clearer, more familiar context for the LLM model. Additionally, the FO mode has generated far more tuples than its Galois counterpart and, consequently, has needed more tokens.



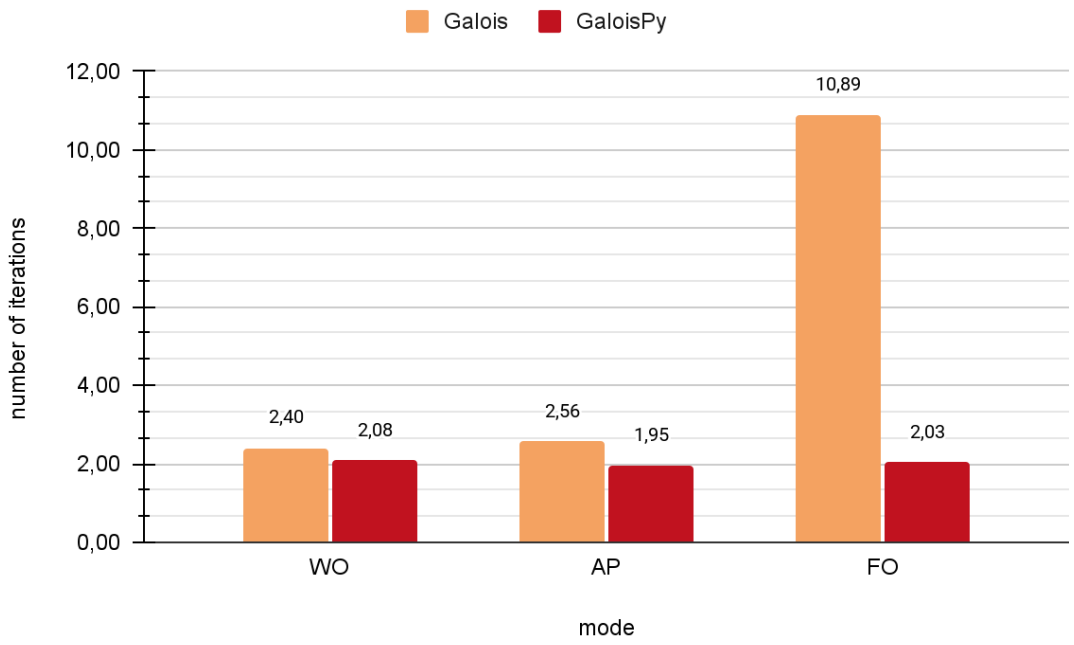
Graph 6 - Number of Tokens metric scores of Galois and GaloisPy over different modes.

The second-to-last metric is the average execution time, which shows that GaloisPy has taken slightly more time per query than Galois, especially for those modalities utilizing the Key-Scan strategy. It is important to note, anyhow, that the second step of the Key-Scan in our re-implementation is not parallelized yet, which could explain the difference in execution times. Nevertheless, GaloisPy has generated more results than Galois, which, naturally, has required more time.



Graph 7 - Average time per query for Galois and GaloisPy.

Finishing with the average number of iterations, as we can see, GaloisPy has completed a query in fewer iterations on average. However, in the case of the FO mode, GaloisPy has needed far fewer iterations, just around a fifth of those required by Galois. This highlights a more efficient system, likely due to a better choice of the physical operator to use, or to better management of the already generated content.



Graph 8 - Average number of iterations per query for Galois and GaloisPy.

In conclusion, our re-implementation, GaloisPy, outperforms the original system in all metrics, except token usage and execution time. The data shows that GaloisPy provides more complete and accurate results, especially regarding the *F1 Cell* and *Tuple Constraint*, which expresses the overall quality of the tuples generated, in a smaller number of iterations, particularly in the FO configuration. However, the excess in token usage in the FO and WO modes suggests that using the *Key-Scan* strategy, coupled with the new memory chat management, requires a considerably higher number of tokens than the other GaloisPy modes; nevertheless, it achieves decidedly worse performance. This highlights the ineffectiveness of such a scan strategy.

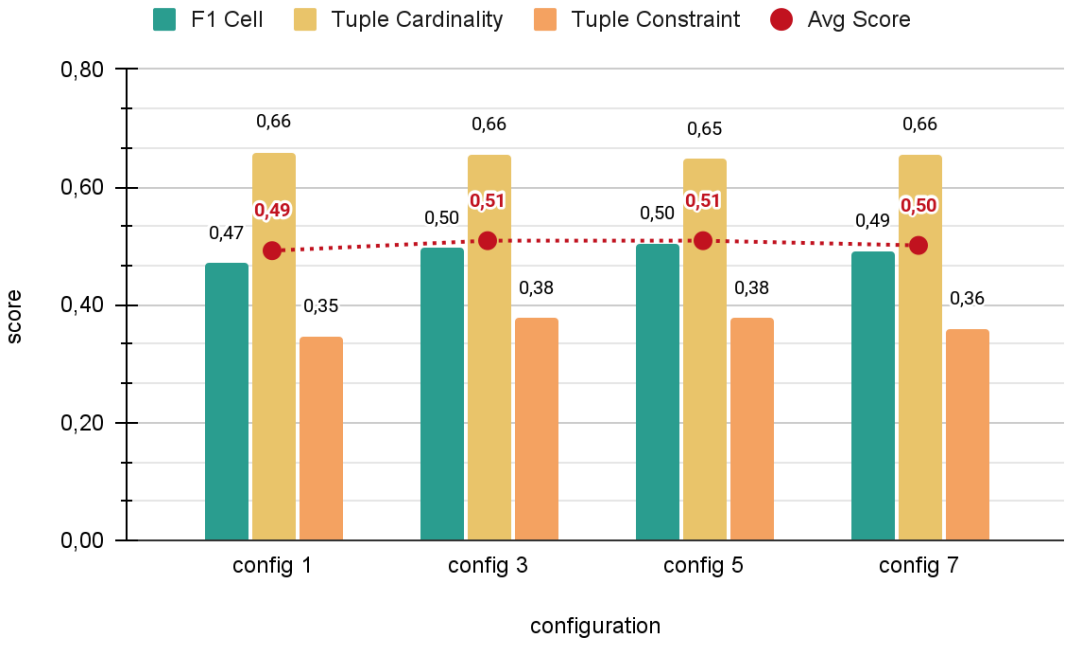
3.3.2 - Performance analysis of the different GaloisPy configurations

In this paragraph, we will analyze the impact of the following improvements that we have developed: the inclusion of additional description fields in the database schema, a retry mechanism in case of empty or already-generated responses, and prompt rewriting from SQL-like prompts to natural language questions. The tested configurations are as follows:

Configuration numb	Description	Retry	Rewrite
config 1	no	no	no
config 2	no	no	yes
config 3	no	yes	no
config 4	no	yes	yes
config 5	yes	no	no
config 6	yes	no	yes
config 7	yes	yes	no
config 8	yes	yes	yes

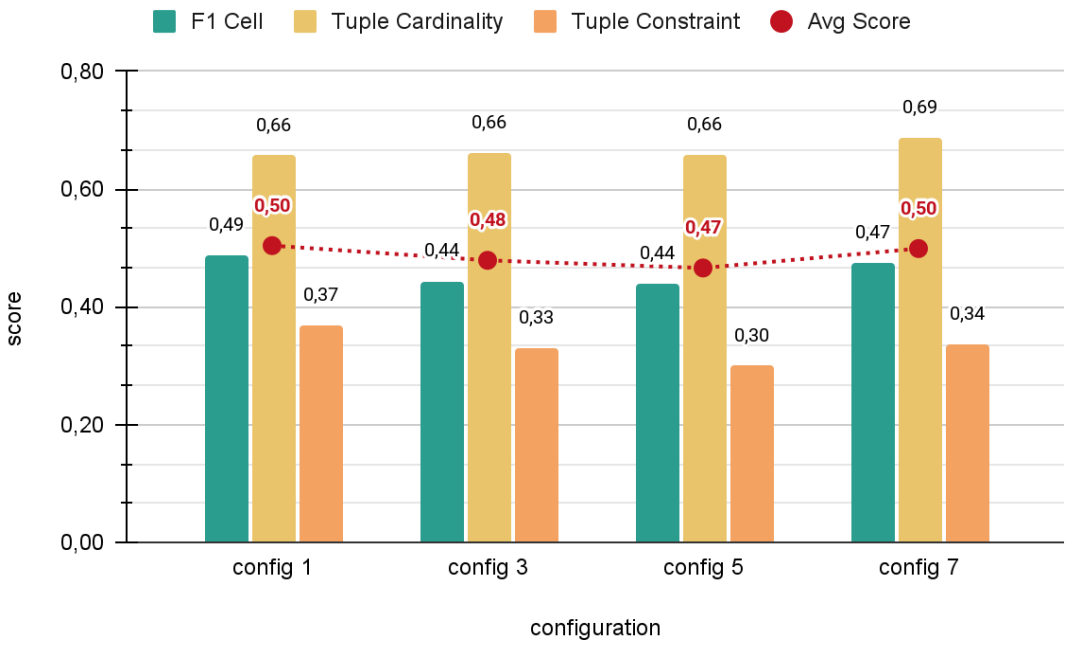
Table 3 - List of all tested configurations with the specified improvements used.

Starting with the NL mode, as we can see from the graph below, the overall performance is almost the same in all configurations, highlighting that neither the introduction of description fields in the database schema nor the introduction of a retry mechanism helps to increase the capabilities of this particular modality. However, the overall scores achieved by the NL mode are quite high, highlighting that natural language prompts are an effective strategy.



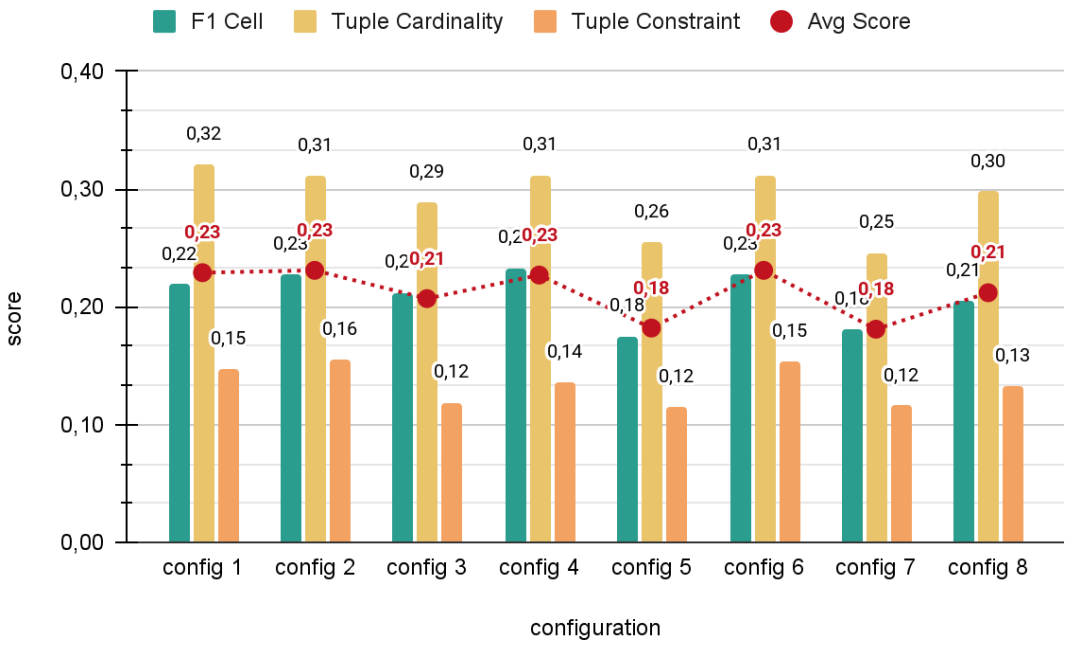
Graph 9 - Metrics values over different NL configurations.

Moving on to the second baseline mode, SQL, we can notice that it is also unaffected by any kind of improvements. However, the performance metrics are comparable to those achieved by the other baseline model. This indicates that querying the LLM model using SQL queries only remains an effective approach.



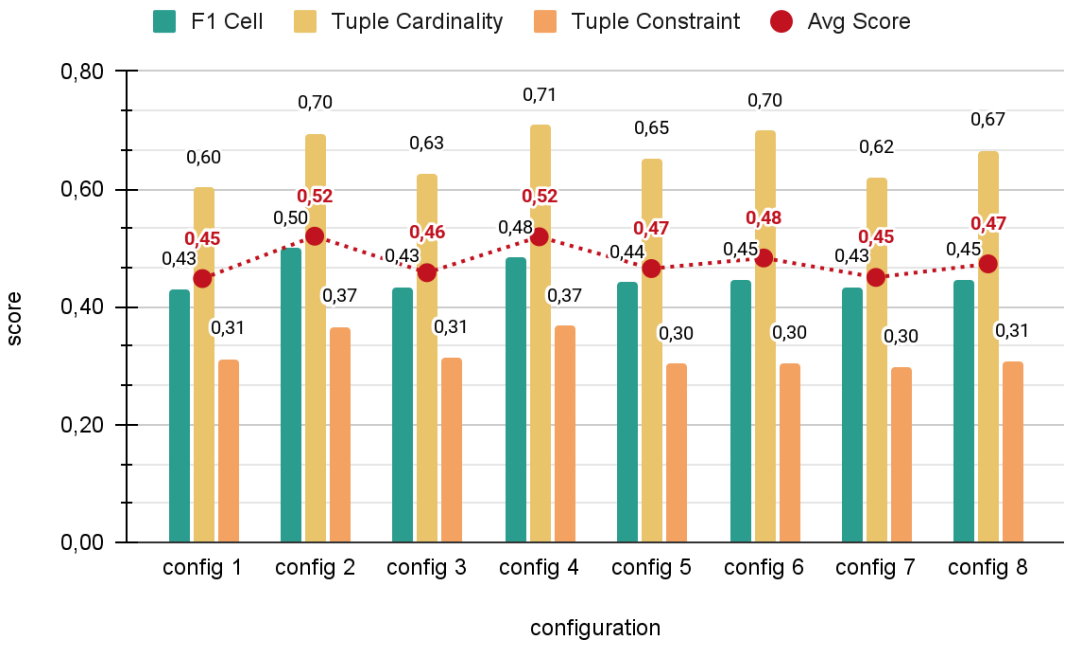
Graph 10 - Metrics values over different SQL configurations.

Analyzing the WO mode's performance, we can see that its overall scores are quite low, particularly with regard to *Tuple Constraint* and *F1-Cell*. This mode seems to benefit a little bit from the prompt rewriting, increasing a bit the values of *Tuple Constraint* and *F1-Cell*, while the introduction of additional description fields seems not to affect the model's capability, but instead it has reduced a little bit its performance. Lastly, the retry strategy, although it was developed to increase the Tuple Cardinality metric, in this case, has achieved the opposite effect, reducing not only such a metric but also all the others.



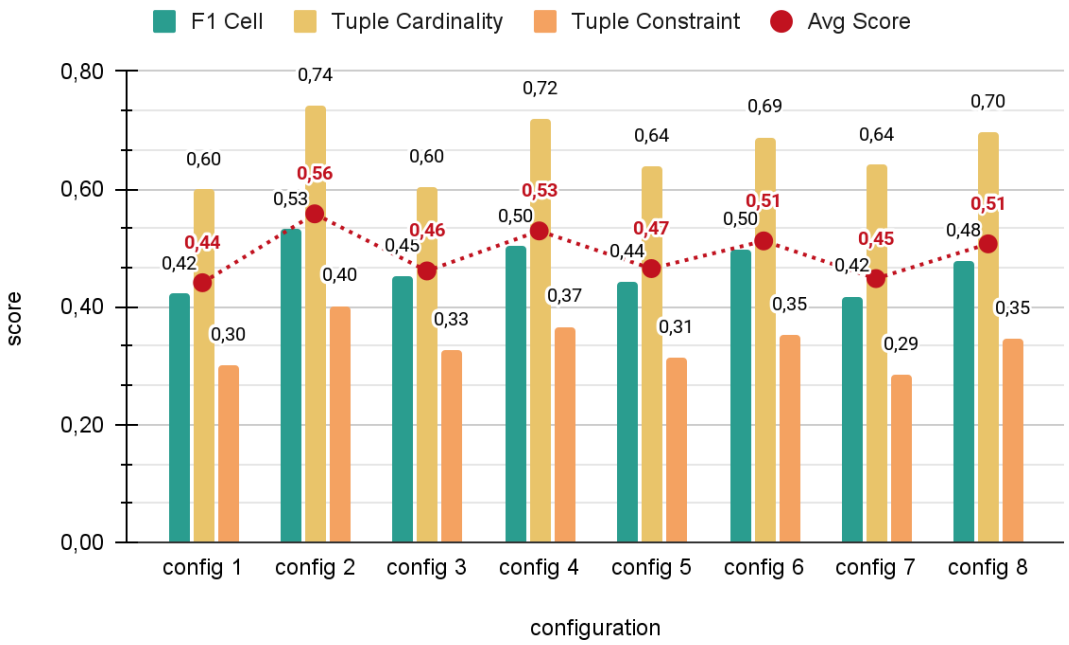
Graph 11 - Metric values over different GaloisPy WO configurations.

Moving on to the AP mode, we can see from the data below that this modality has benefited significantly from the prompt rewriting, especially in terms of *Tuple Cardinality* and *Average Score*. However, as shown in the data below, this improvement performs better when used without the additional descriptions; indeed, if we compare configuration 2 with configuration 6, we can notice that both *F1-Cell* and *Tuple Constraint* have decreased. Regarding the retry strategy, its benefit has been marginal, both when it has been used alone or in combination with the other improvements.



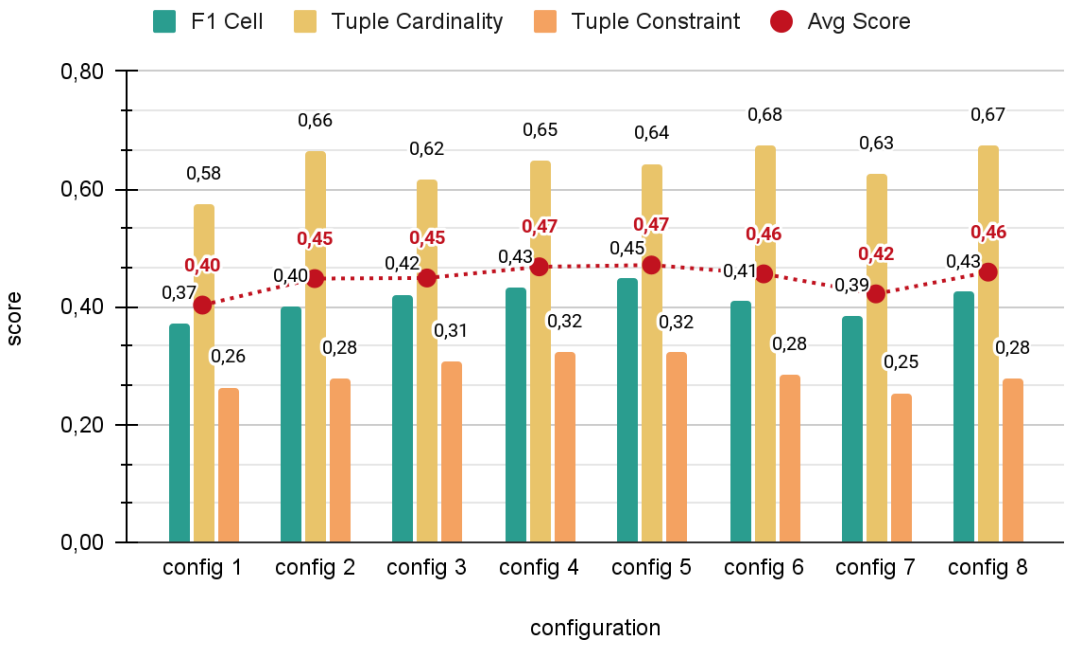
Graph 12 - Metric values over different GaloisPy AP configurations.

Second-to-last mode, SP has achieved the highest performance among all the modalities tested, achieving an average score of 0,56, overcoming Galois SP with *Llama 3.1 70b*. As shown in the graph below, this mode has benefited significantly from the prompt rewriting strategy, improving all metrics, in particular, *Tuple Constraint* and *Tuple Cardinality*. As in the AP mode, this improvement has worked better when not combined with others. It is interesting to note that SP has performed similarly, but a little better than AP, showing that the logical optimization helps the model in achieving better performance, but the impact has been lower than expected.



Graph 13 - Metric values over different GaloisPy SP configurations.

Last on the list is FO mode, which has behaved differently from the other GaloisPy modalities. As the data below shows, it has performed better in terms of *Tuple Constraint* and *F1 Cell* when prompt rewriting and retry were both used. Furthermore, FO mode seems to have benefited from the additional description fields, which increased its performance slightly in terms of *F1 Cell* and *Tuple Cardinality* compared to configurations 5 and 8, which did not use such an improvement. However, the *Tuple Constraint* has decreased slightly compared to configurations that did not use the description fields.



Graph 14 - Metric values over different GaloisPy FO configurations.

In conclusion, from the data we have collected from these tests, we can infer that, first of all, the prompt rewriting has been the most effective improvement strategy we have developed, especially for the AP and the SP mode, in which both have surpassed the score of 0,5, matching or even overtaking more powerful models tested in the original paper, such as *Llama 3.1 8b* (maximum score 0,528) or *GPT-4o mini* (maximum score 0,468). However, it is interesting to note that this improvement has been more efficient in modalities that use the *Table-Scan* retrieval strategy, rather than in others, which use the *Key-Scan*. Secondly, the retry mechanism has not worked as expected, worsening or not changing the performance in almost any mode. Although this enhancement aims to improve the *Tuple Cardinality* by overcoming temporary empty responses, in many cases, it has achieved the opposite effect, reducing or matching the same cardinality, but lowering other metrics such as *F1 Cell* and *Tuple Constraint*. Lastly, the additional description fields have not helped the model to better understand the context, leading to performance issues. Indeed, in almost all cases, the usage of the descriptions lowers the performance compared to not using them, except for the FO mode, in which it seems to be slightly effective, producing a slightly higher average score among the configurations that have adopted it. It is interesting to note that these additional descriptions seem not to

have had any noticeable effect on the two baseline modes, suggesting that for these two modalities, a more detailed description of the context was not necessary.

However, it is important to underline that, although we have set the model temperature to 0 and the top-p to 0,01, we were not able to get a full reproducibility of the results, suggesting that even the smallest difference in the hardware architecture or the presence of non-editable and hidden parameters of the LLM model can differ a bit the results between different runs, as it can be seen by the NL and SQL experiments. Nevertheless, repeating the same experiments several times, we have seen that often, we got the same results, showing in any case a good level of reproducibility.

Chapter 4: Retrieval-augmented generation experiments

In this second round of experiments, we have evaluated the RAG capabilities of the GaloisPy system. To objectively analyze its performance, we have taken, as in the previous round of tests, the same test suite of Galois. Then, we have compared the strengths and weaknesses of both implementations.

In this chapter, we will present the experimental setup, covering the datasets used and their content. We then present the results obtained with GaloisPy, followed by a comparative analysis of both systems, focusing on the specific improvements, potential trade-offs, and the overall effectiveness of the optimization methods introduced in our version.

4.1 - Experimental setup

For these experiments, we have used two additional AI models, one for embeddings and one for reranking. These are:

- *mxbai-embed-large:* Developed by Mixedbread, it is a small yet powerful embedding model that outperforms other commercial models, such as OpenAI's *text-embedding-3-large*.
- *mxbai-rerank-xsmall-v1*: A reranking model from Mixedbread. It is the smallest model of their lineup, but it offers a good balance between size and performance. In some benchmarks, it scores similarly to larger models such as BAAI's *bge-reranker-large*.

Model	Producer	Size	Context window	Distributed by
mxbai-embed-large	Mixedbread	0.3B	512	Ollama
mxbai-rerank-xsmall-v1	Mixedbread	70.8M	512	Hugging Face

Table 4 - RAG models' specifications.

Then, we have taken the same two datasets used in the original experiments, which are:

- *Fortune:* A dataset that includes information about the 2024 Fortune 500[7] companies. It includes information such as ranking position, company name, operating sector, number of employees, and more. It includes around 500 documents, all in markdown format, split into chunks of 400 tokens each, with 100 tokens for overlap.
- *Premier:* It contains data from the first six match-days of the 2024-2025 Premier League season, scraped from BBC News. It includes information about Arsenal matches, man-of-the-match awards, goals scored, and much more. This dataset consists of almost 60 news articles, all in markdown format. Its documents are split into chunks of 128 tokens, with 32 for the overlap.

Dataset name	Dataset source	# of queries	Avg. expected cells	Experiment type
Premier	BBC News	5	57.8	RAG
Fortune	Kaggle	10	7.9	RAG

Table 5 - Statistics about RAG datasets.

At each query, we have retrieved the top 100 documents from the *Chroma* vector database to serve as context to the LLM model, including them in the system prompt. In case of reranking, we have retrieved 300 documents instead, and after the re-ordering, we have selected the top 100.

Regarding the LLM model, we have used *GPT-4.1 nano* again, with the same configuration as for the IK experiments.

Model	GPT-4.1 nano
Temperature	0.0
Top-p	0.01
Frequency penalty	0.0
Presence penalty	0.0
Max token	13107
Content window size	1000000
Tokenizer (*tiktoken*)	cl100k_base
Physical opt threshold	0.8

Table 6 - Model configuration.

Finishing with the modalities tested, as in the IK experiments, we have kept the same two baseline modes, but we have tested GaloisPy only on two modalities: AP and FO.

[7] It is an annual list compiled and published by Fortune magazine that ranks 500 of the largest United States corporations by total revenue for their respective fiscal years.

We have not considered the other modalities to be more aligned with the original experiments.

4.2 - Evaluation metrics

To evaluate the performance of the system, we kept the same metrics used by the Galois team in the original paper, which are:

- *Avg Score:* The average of *F1 Cell*, *Tuple Cardinality*, and *Tuple Constraint* metrics. It measures the overall quality of the result set.

$$Avg\ Score = \frac{F1\ Cell\ +\ Tuple\ Cardinality\ +\ Tuple\ Constraint}{3}$$

- *Number of tokens:* The total number of tokens used to execute all the experiments' queries.

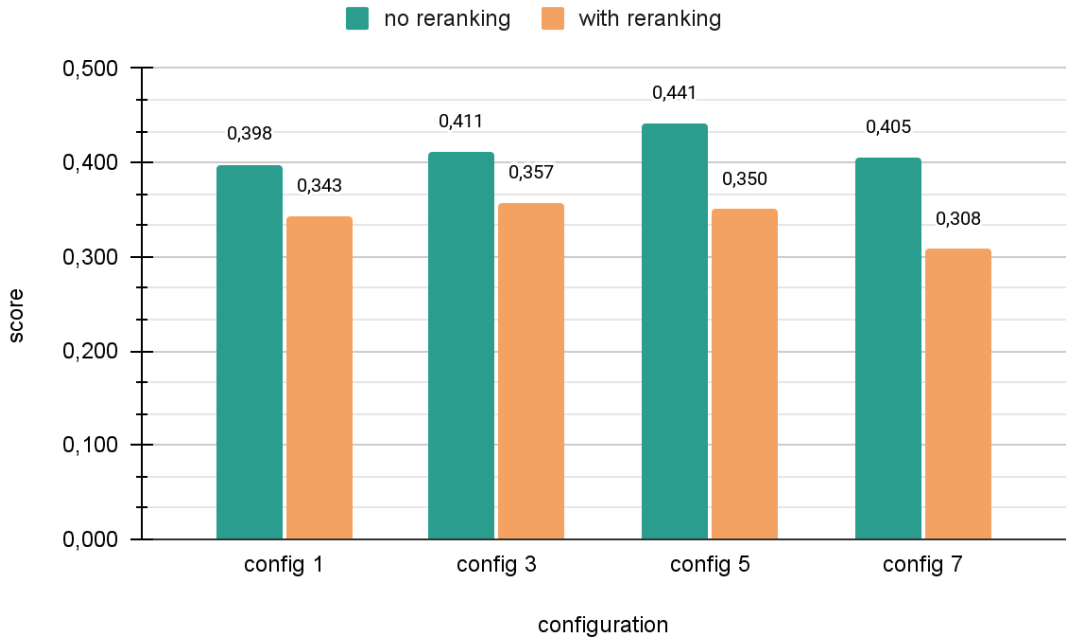
4.3 - Results and comparisons

4.3.1 - Galois vs GaloisPy

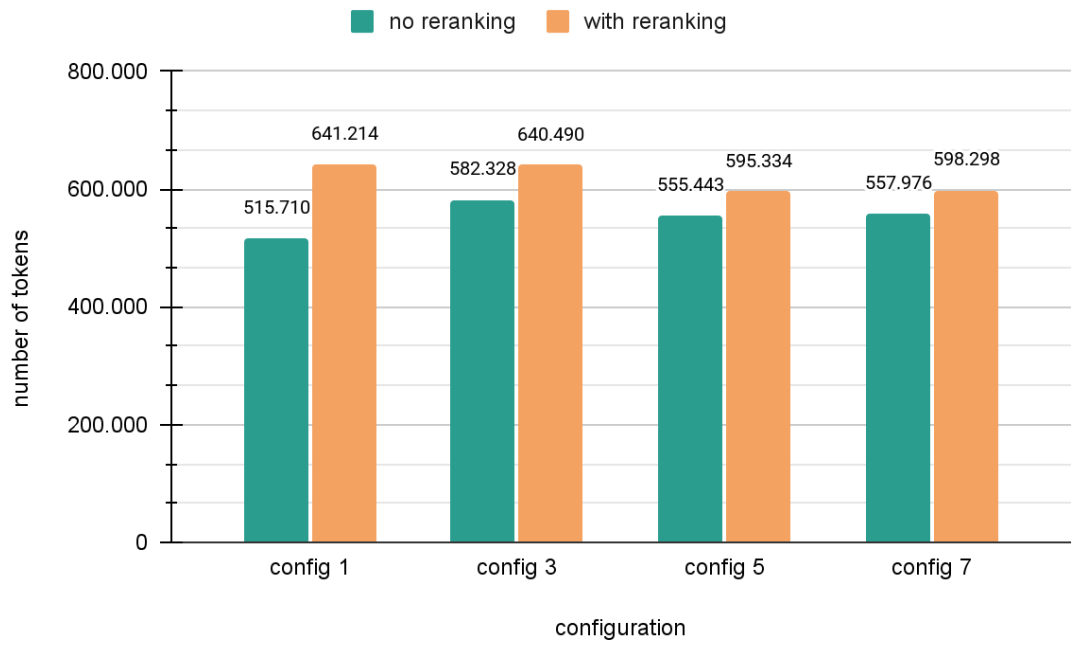
As we did before, we have tried to adjust the Galois code to run the experiments with our configuration, thus the *GPT-4.1 nano* as LLM model and the *mxbai-embed-large* embedding model. This time, unfortunately, we were unable to run such experiments due to the impossibility of creating the vector database for Chroma, in which to store document chunks, using the embedding model through Ollama. To overcome this, we needed to deeply rewrite all the RAG sections, but, due to its hardness, we opted not to proceed further. For this reason, a true comparison between the original system and our re-implementation is not present; however, we suppose that its performance could be around the same as the ones achieved during the IK experiments or a bit lower, due to the size of the LLM used.

4.3.2 - Performance analysis of the different GaloisPy configurations

Starting with the NL model, as we can see from the graph below, the achieved scores are quite good, especially for the size of the NLP model and the number of documents provided. As we can notice, the introduction of the reranker strategy has increased the performance a bit, while the additional description fields have worked better alone rather than in combination with other improvements. Regarding the reranker, its introduction has decreased the performance noticeably, especially if the additional descriptions were used, while the token usage has increased, especially in configuration number 1, where it has used over 100 thousand more tokens.

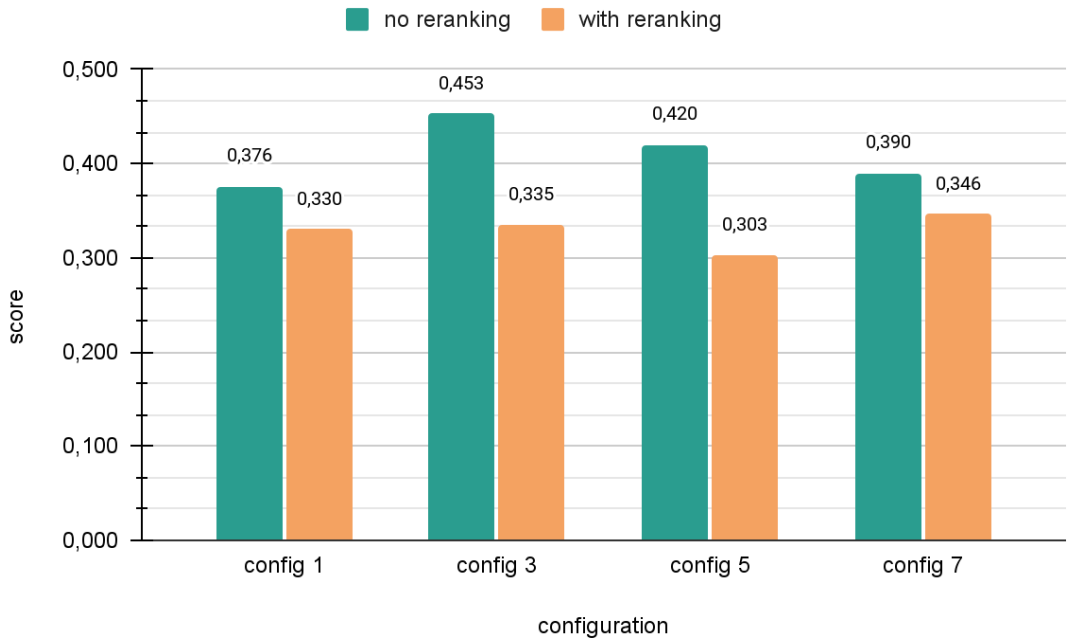


Graph 15 - Average Score with and without reranking for NL.

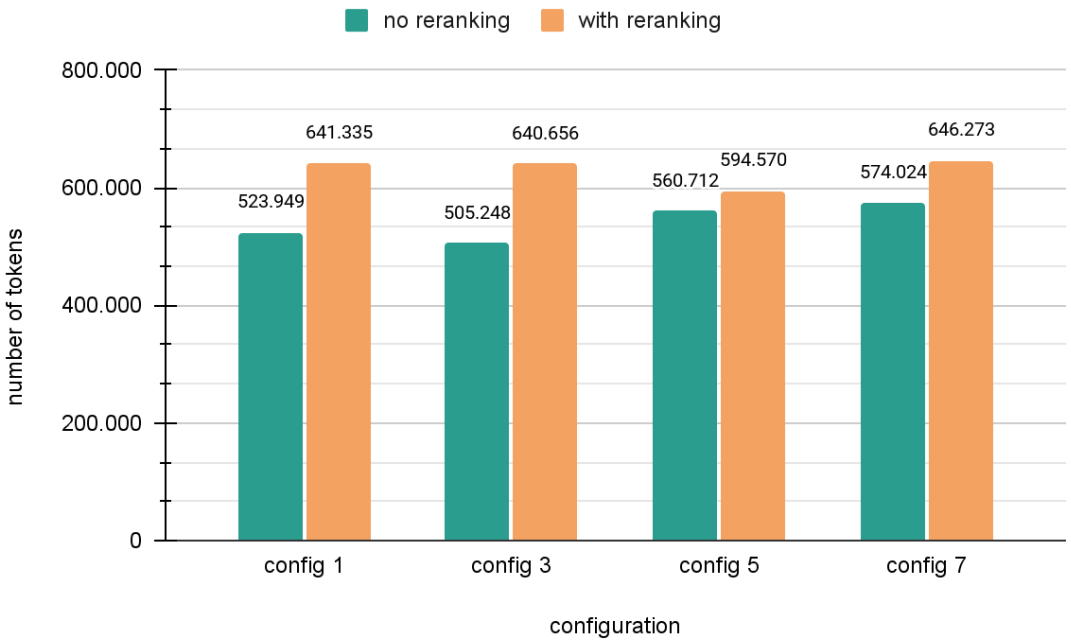


Graph 16 - Number of tokens used by NL with and without reranking.

Moving on to the SQL mode, its general trend is similar to the previous one. Also, in this case, the introduction of the reranker has worsened the overall performance, especially in configurations where additional description fields were not used. However, unlike the NL mode, the difference between with and without the reranker is noticeably higher, especially in configurations 3 and 5. Moreover, the number of tokens used by the model without reranking is visibly lower, especially in configurations 1 and 3. This highlights once more the ineffectiveness of such a strategy.



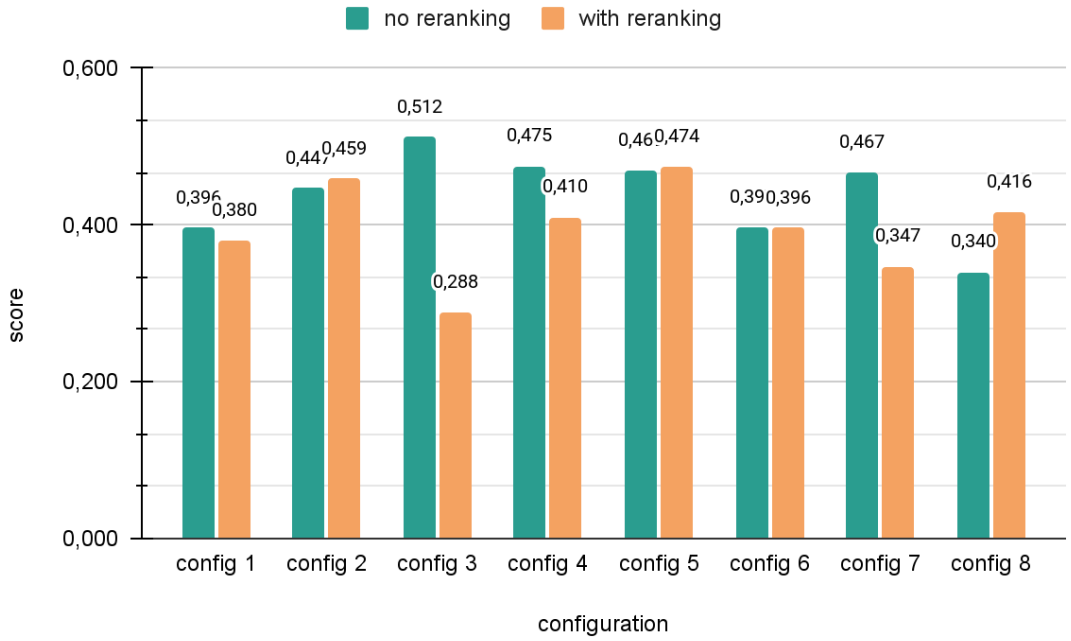
Graph 17 - Average Score with and without reranking for SQL.



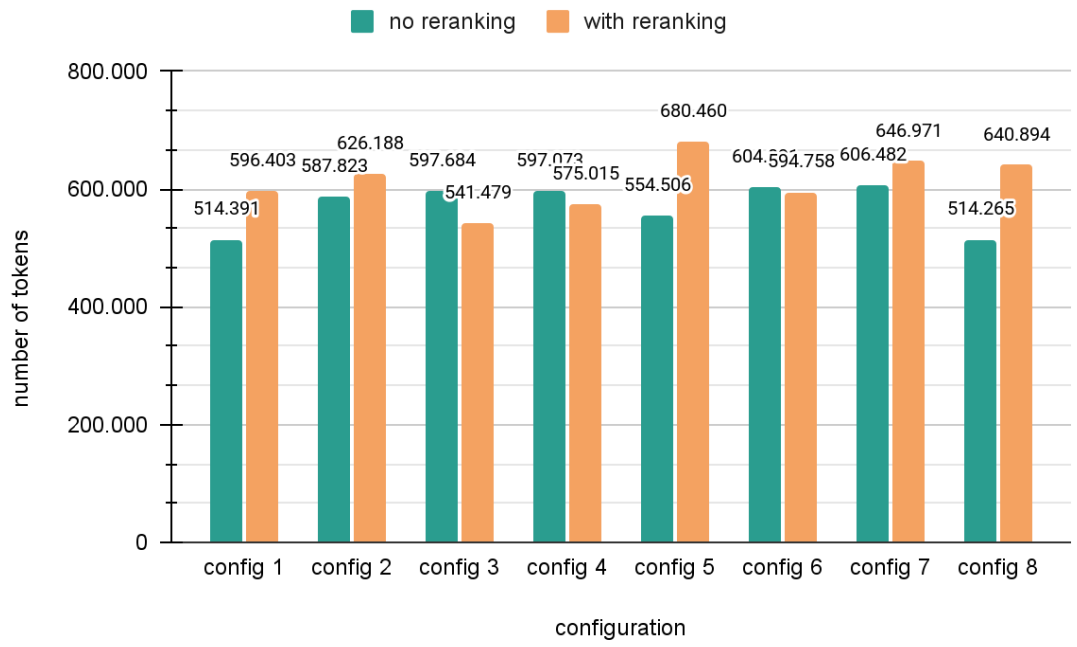
Graph 18 - Number of tokens used by SQL with and without reranking.

The second-to-last modality tested is AP. In this mode, the retry strategy has improved performance, especially when no other enhancements were applied. This pattern differs from what was observed during the IK experiments, indicating that encouraging the model to generate more tuples is more effective than using a more polished prompt,

probably because it already has all the information needed to answer the question. Additionally, the reranker model has performed worse when additional descriptions were not used, but has matched or improved performance in almost all configurations that included these descriptions. Regarding token usage, it is generally observed that when the reranker model has been used, the number of tokens has increased, as seen in previous modes. However, in some configurations, such as 3 and 4, the number of tokens used has decreased slightly.

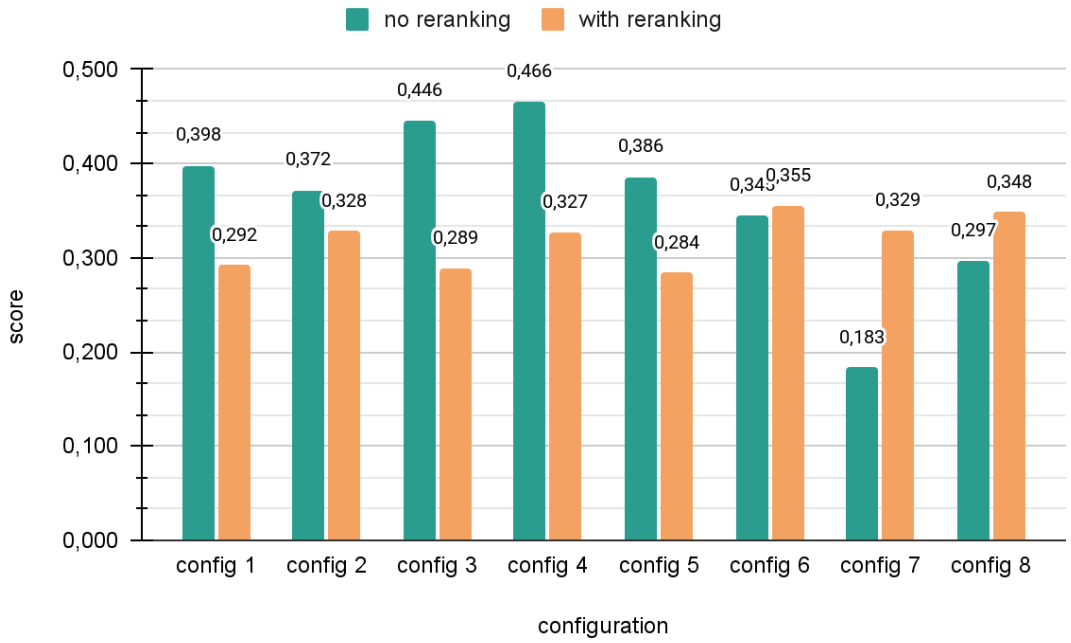


Graph 19 - Average Score with and without reranking for GaloisPy AP.

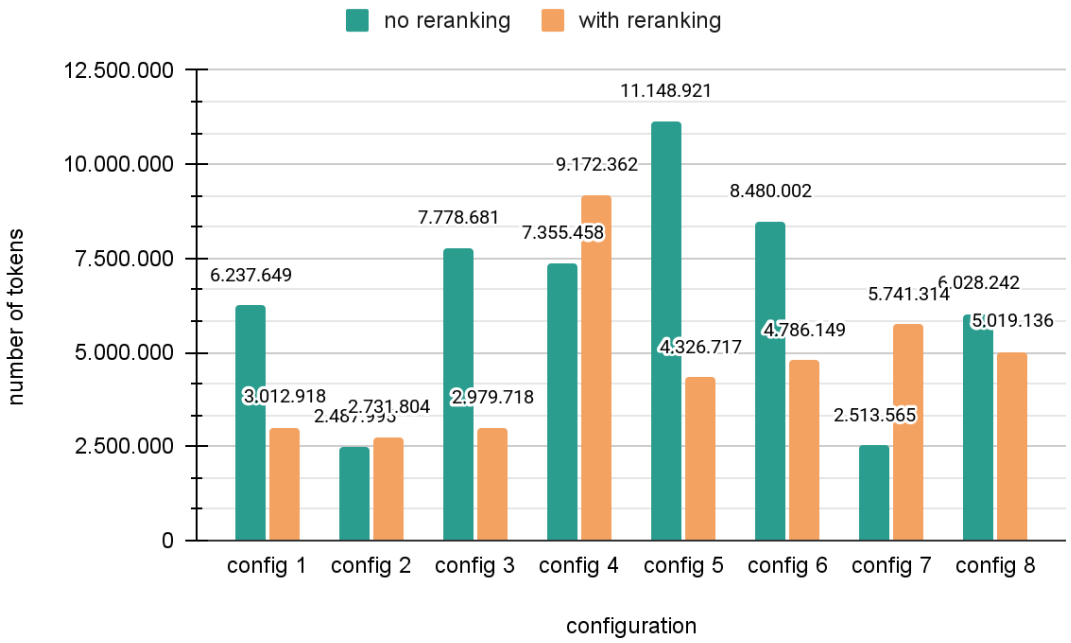


Graph 20 - Number of tokens used by GaloisPy AP with and without reranking.

Finishing with the FO mode, we can see that it has performed significantly better without the additional description fields and with the retry strategy enabled. This trend is similar to what we have seen before with the AP mode, where encouraging the LLM model to produce more results can effectively improve the performance more than providing clearer prompts or a usage scenario. Nevertheless, providing the model with a more accurate and detailed context has decreased the performance noticeably, especially when coupled with prompt rewriting. Moving to the reranker model, the results achieved are the opposite of what we have just seen. In this case, the FO mode has performed better when both the additional descriptions and the prompt rewriting were used. However, the overall quality of the results achieved was lower than the ones obtained before, highlighting the ineffectiveness of such a model again. Lastly, regarding the token usage, we can notice that this mode has used far more tokens than the others, but, interestingly, the introduction of the reranker model has significantly decreased the number of tokens used, in particular in configurations 3, 5, and 6. Anyhow, the performance achieved in such configurations was lower than that achieved without the reranker, except for configuration 6, justifying partially such a decrease.



Graph 21 - Average Score with and without reranking for GaloisPy FO.



Graph 22 - Number of tokens used by GaloisPy FO with and without reranking.

In conclusion, based on our improvements, we found that the most effective was the retry strategy. This indicates that encouraging the model to produce more tuples can significantly boost overall performance, probably because the model already has all the necessary information to generate the answer. Additionally, from the experimental

results, we see that GaloisPy outperformed the two baseline modes, demonstrating that the GaloisPy strategy can be successfully applied to RAG technology. However, the performance achieved was slightly below our expectations, likely due to several factors.

First, for these tests, we used *GPT-4.1 nano*, a budget-friendly model that balances performance and cost per million tokens; however, its reasoning capabilities are noticeably lower compared to *Llama 3.1 70b*, used in the original paper. This, combined with the large number of documents injected into the system prompt to provide context, could be a challenge for such a small model. The second factor, a direct result of the first, is the so-called *"Lost-in-the-Middle"* effect. Providing a large amount of text in a single prompt to a small model can cause difficulties in retrieving relevant information, leading to a focus on either the top or the bottom of the prompt. In these tests, we pass the model several thousand tokens as context, and small models like *GPT-4.1 nano* may struggle to detect and analyze relevant information. Reducing the number of retrieved documents to 10 or 25 might greatly improve overall performance. Lastly, the chunk size is objectively too small, especially in *rag-premier*, where each chunk contains 128 tokens, with 32 for overlap. Such a small context can reduce the efficiency of the embedding model because each chunk does not contain enough content. Increasing these numbers to 1,000 or more, while reducing the number of retrieved documents, could significantly enhance overall performance.

Moving to the reranking model, we can notice that it has led to the opposite effect than we expected. In almost all cases, the reranker has sensibly reduced performance while noticeably increasing the token usage. Although on benchmarks the performances achieved by the reranking model are remarkable, the overall capabilities of detecting semantic correlations between query and fragments can be lower than the capabilities of the embedding model, leading to poor classification, and yet to worse results. While a more robust reranking model would likely yield the expected improvements, hardware limitations of the testing environment precluded the use of larger models, restricting the choice of reranking model specifically to the *mxbai-rerank-xsmall-v1*.

Chapter 5: The Nobel Prize experiment

Additionally, we have conducted another experiment in which we have evaluated our system, GaloisPy, on a new dataset about the Nobel Prizes, a different topic not covered in the original tests. However, the purpose of this test is not only to assess GaloisPy's performance on a new subject, but also to understand how our system responds to different filtering conditions. The idea is to give the program a series of queries with a common structure or template, where only the filtering conditions vary.

In this chapter, we will present the experimental setup, covering the datasets used and their content. We then present the results obtained with GaloisPy, focusing on how the system reacts to the specific templates, also highlighting which modes and configurations have performed better than the others and in which scenarios.

5.1 - Experimental setup

The dataset, named *"Nobel Prize Winners Dataset (1901-2025)"*, distributed on Kaggle by Ahmed Mohamed Zaki, contains data regarding the Nobel Prize awardees from 1901, the year in which it was established, to 2025. Since *GPT-4.1* and all its variants were released in April 2025, and its knowledge cutoff[8] is estimated to be around June 2024, we exclude from the dataset records about 2024 and 2025 awardees.

Since the dataset is made by just one single table containing many different fields, we have decided to split it into two distinct tables, in which the first includes data about awardees, such as award year and date, the Nobel Prize's field, the winner's name, and more; meanwhile, the other reports information about the winner, like, for instance, birth date and place, the citizenship, and, eventually, death date and place.

Dataset name	Dataset source	# of queries	Avg. expected cells	Experiment type
nobel-prizes	Kaggle	50	10.4	IK

Table 7 - Statistics about the Nobel Prizes dataset.

[8] It is the point in time beyond which a model has not been trained on new data. Any information about events after this date is absent from the model's training data.

As before, we have evaluated GaloisPy's performance over the same set of metrics, keeping the exact similarity mechanism used in the previous experiments.

5.2 - Results and comparisons

As we did in the previous experiments, we have tested GaloisPy in all its configurations, which are the following:

Configuration numb	Description	Retry	Rewrite
config 1	no	no	no
config 2	no	no	yes
config 3	no	yes	no
config 4	no	yes	yes
config 5	yes	no	no
config 6	yes	no	yes
config 7	yes	yes	no
config 8	yes	yes	yes

Table 8 - List of all tested configurations with the specified enhancements used

Moreover, to better deliver and analyze how different modalities and configurations have behaved, we will present one template at a time.

5.2.1 - Template number 1

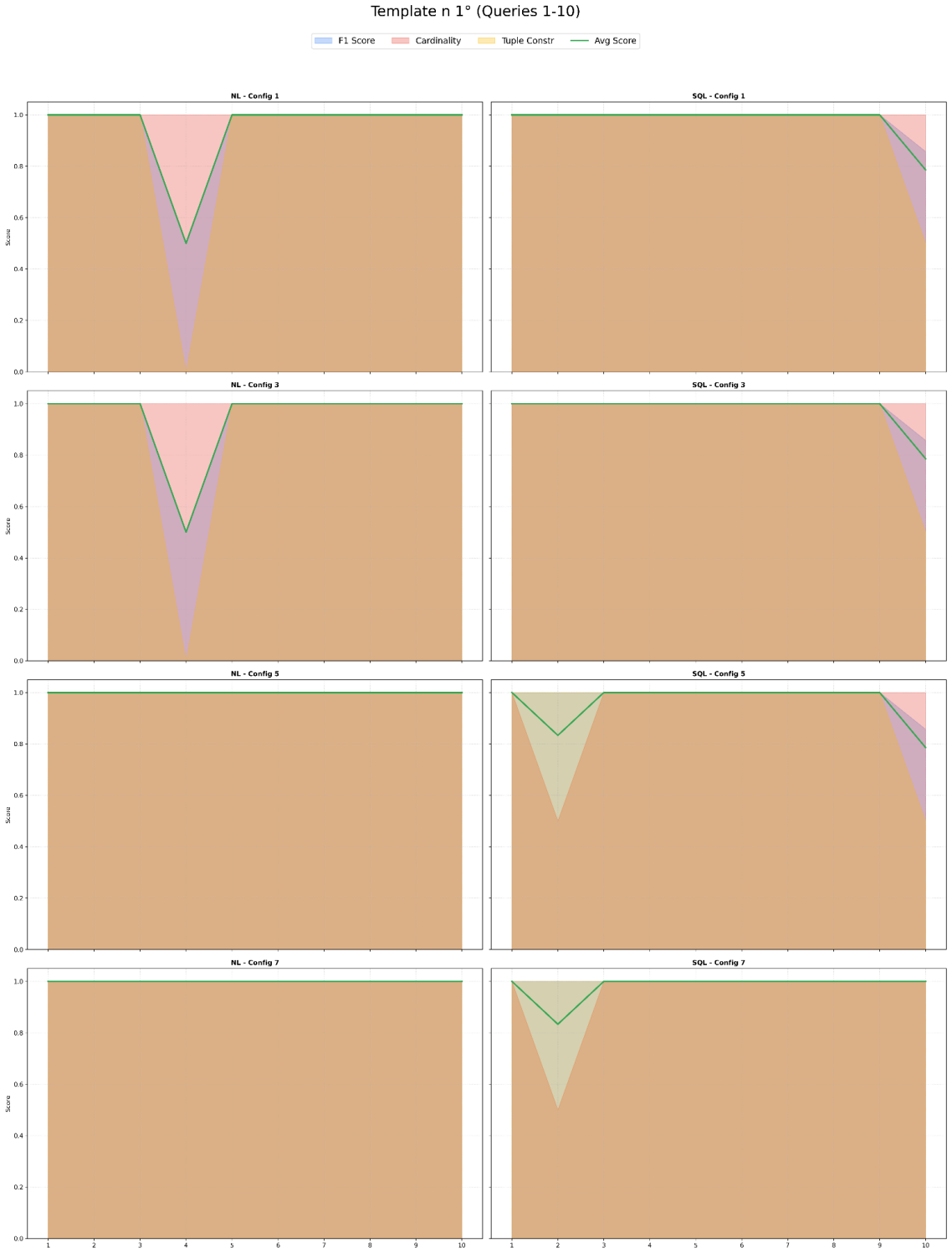
In the first template, we instruct the system to provide the award year and prize category of some of the most famous Nobel Prize winners, including Albert Einstein, Niels Bohr, and Barack H. Obama. The winner's name is specified in the WHERE clause.

```
SELECT award_year, category
FROM nobel_prizes_1901_2023_awardees
WHERE name = 'Albert Einstein';
```

Figure 17 - Template number 1 with filtering condition over the name of the winner (in this example, Albert Einstein).

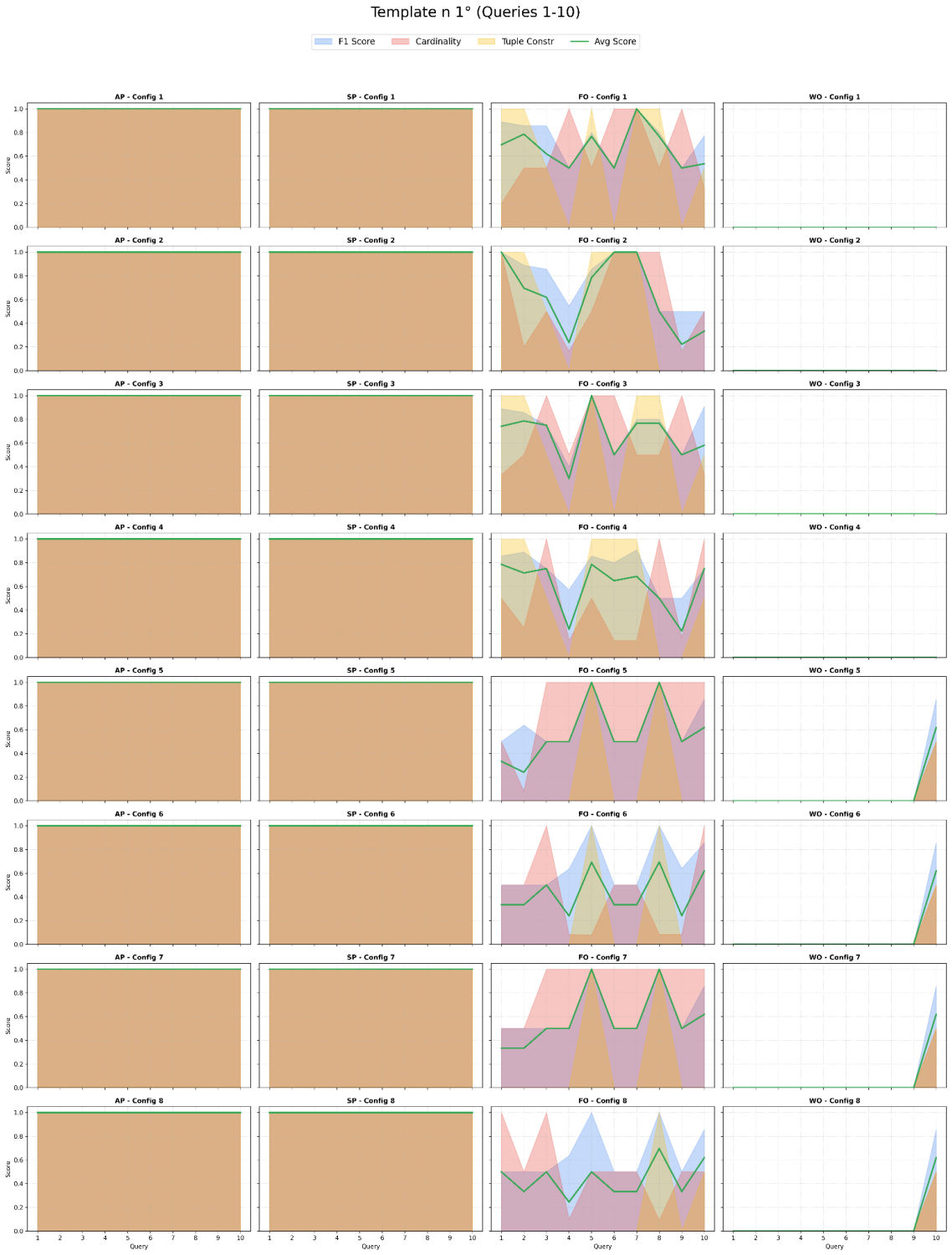
Starting from the two baselines, as we can see from the graphs below, the NL mode has performed better compared to the SQL model with the additional description fields, achieving the highest accuracy. However, if such additional descriptions were missing, it performed a bit worse than the other baseline mode, especially on query 4, where it

achieved 0,0 in *Tuple Constraint*. All in all, the performance in this first template for both the baseline modes is still good.



Moving to GaloisPy, we can see that both AP and SP modes have perfectly answered all the queries in all the different configurations, while, on the other hand, the WO mode

often has not provided any results. However, when the additional descriptions were used, they produced some results sporadically, but they are limited only to query number 10 and, in these cases, they achieved a modest score of around 0,6. In any case, the performance of WO is very poor. Regarding the FO mode, its achievements were very fluctuating, but it has performed generally better without additional description fields and with the prompt rewriting strategy enabled.



In conclusion, the AP and SP modes demonstrated superior performance compared to the other GaloisPy modes, as well as the NL and SQL baseline modes. This demonstrates the effectiveness of the Table-Scan strategy when coupled with the inclusion of the winner's name, which is a crucial hint for the retrieval phase.

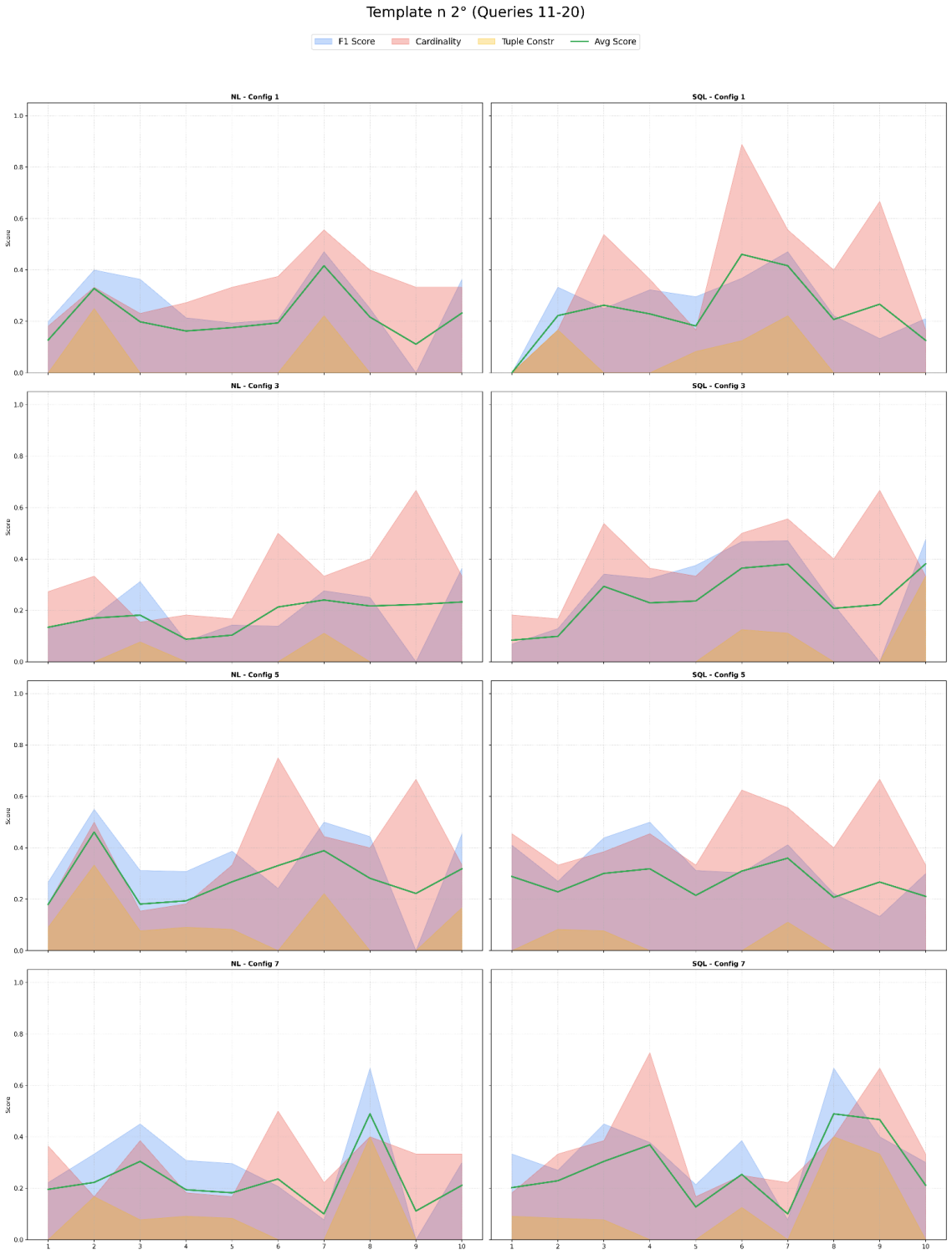
5.2.2 - Template number 2

The second template analyzes the system's ability to retrieve information about Nobel Prizes won in different years. These years range from recent ones, such as 2023 and 2020, to older ones, such as 1901 and 1914. We instruct GaloisPy to retrieve details such as the prize category, the winners' names, and their country of birth. As before, the award year is expressed in the WHERE clause.

```
SELECT a.category, a.name, w.birth_country
FROM nobel_prizes_1901_2023_awardees AS a
  JOIN nobel_prizes_1901_2023_recipients as w
  ON a.name = w.name
WHERE a.award_year = 2023;
```

Figure 18 - Template number 2 with filtering condition over the award year (in this example, 2023).

Beginning with the two baseline modes, as we can notice, the SQL mode has performed better than the NL mode in almost any configuration, confirming its superiority in this template. Although its *Average Score* is generally higher, its *Tuple Constraint* values are comparable to the ones achieved by the NL mode, suggesting that even if it produces more results with a higher accuracy at the cell level, if we take into consideration the whole tuple, the gap between these two modes is reduced and, in some configurations, even surpassed by the NL mode.



Moving to GaloisPy, as we can see from the graphs below, the general performance among all the modes and configurations was really poor. From the results of this experiment, we can see that both AP and SP have sporadically provided relevant results, with the AP mode performing a little bit better. Differently is the case of the FO mode, in

which almost no results were provided, except for configurations 5 and 7, in which it has returned some tuples in all the queries, but with a poor overall quality. Concluding with the WO mode, also in this template, this particular mode has not produced any relevant results, showing a certain difficulty in generating relevant results again.

In conclusion, the two baseline modes have performed noticeably better than GaloisPy's modes in this experiment. They have provided more results with greater consistency and superior quality. GaloisPy's modes, on the other hand, had difficulty furnishing relevant tuples, likely due to the JOIN operation emptying the results produced during the two scanning phases, due to the set intersection operation.

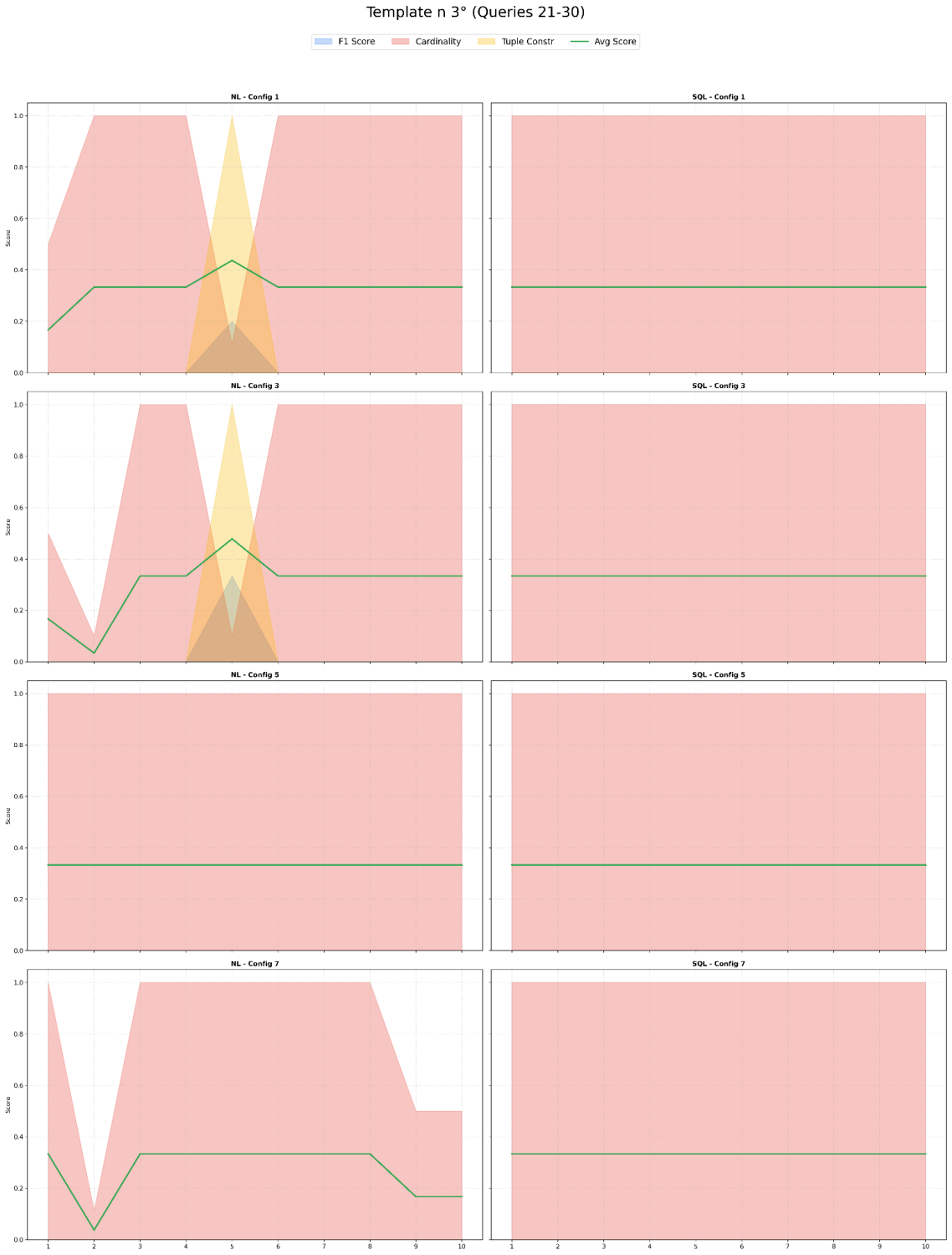
5.2.3 - Template number 3

In the third template, we instruct GaloisPy to count the number of Nobel Prizes won by a specific country, such as Italy, the United States, or the United Kingdom. As usual, the country name is included in the WHERE clause. This time, we push the boundary a bit higher by testing GaloisPy's ability to enumerate tuples.

```
SELECT COUNT(*) AS italian_winners
FROM nobel_prizes_1901_2023_recipients
WHERE birth_country = 'Italy';
```

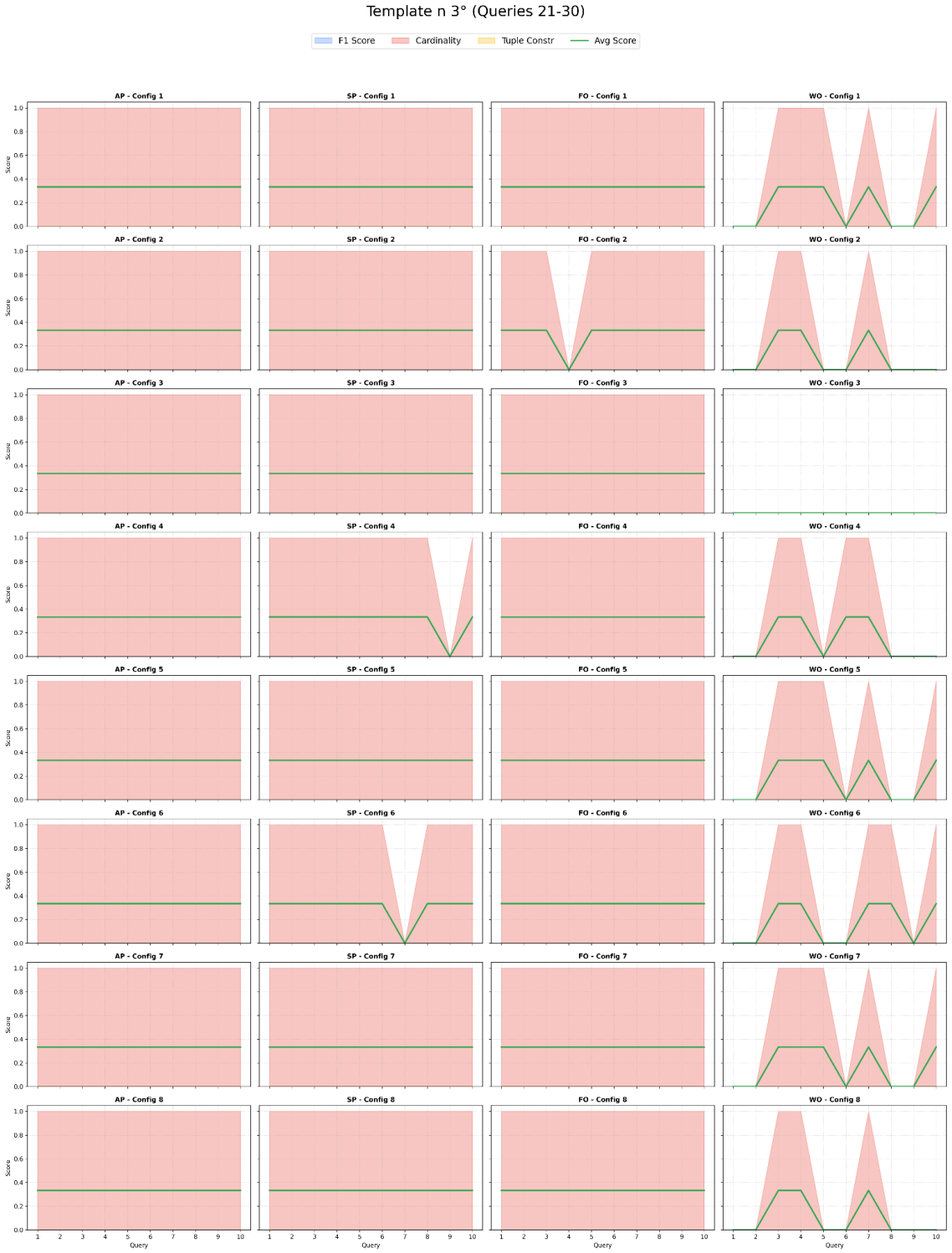
Figure 19 - Template 3 with filtering condition over the country of birth (in this case, Italy).

Starting from the baselines, neither has produced any relevant results in most cases, as indicated by the *F1 Cell* and the *Tuple Constraints*, which are both zero in all configurations. The only exceptions were configurations 1 and 3, in which the NL mode generated relevant results for query 5. In general, however, both baseline modes performed poorly, showing noticeable difficulty with such queries.



Analyzing GaloisPy's modes, the situation does not change. Even there, all the models have struggled in retrieving relevant tuples, with the AP, SP, and FO modes that have performed almost the same, in which only the cardinality was matched. Regarding the

WO mode, its performance has increased compared to previous templates, but its achievements are still poor, showing once again all the limitations of such a mode.



In conclusion, in this particular template, none of the models tested has produced any relevant results, showing difficulties in this kind of query. Indeed, from the results we

got, we can state that, independently of the modality and the configuration used, this LLM model has struggled to produce relevant results, probably due to its excessively small size.

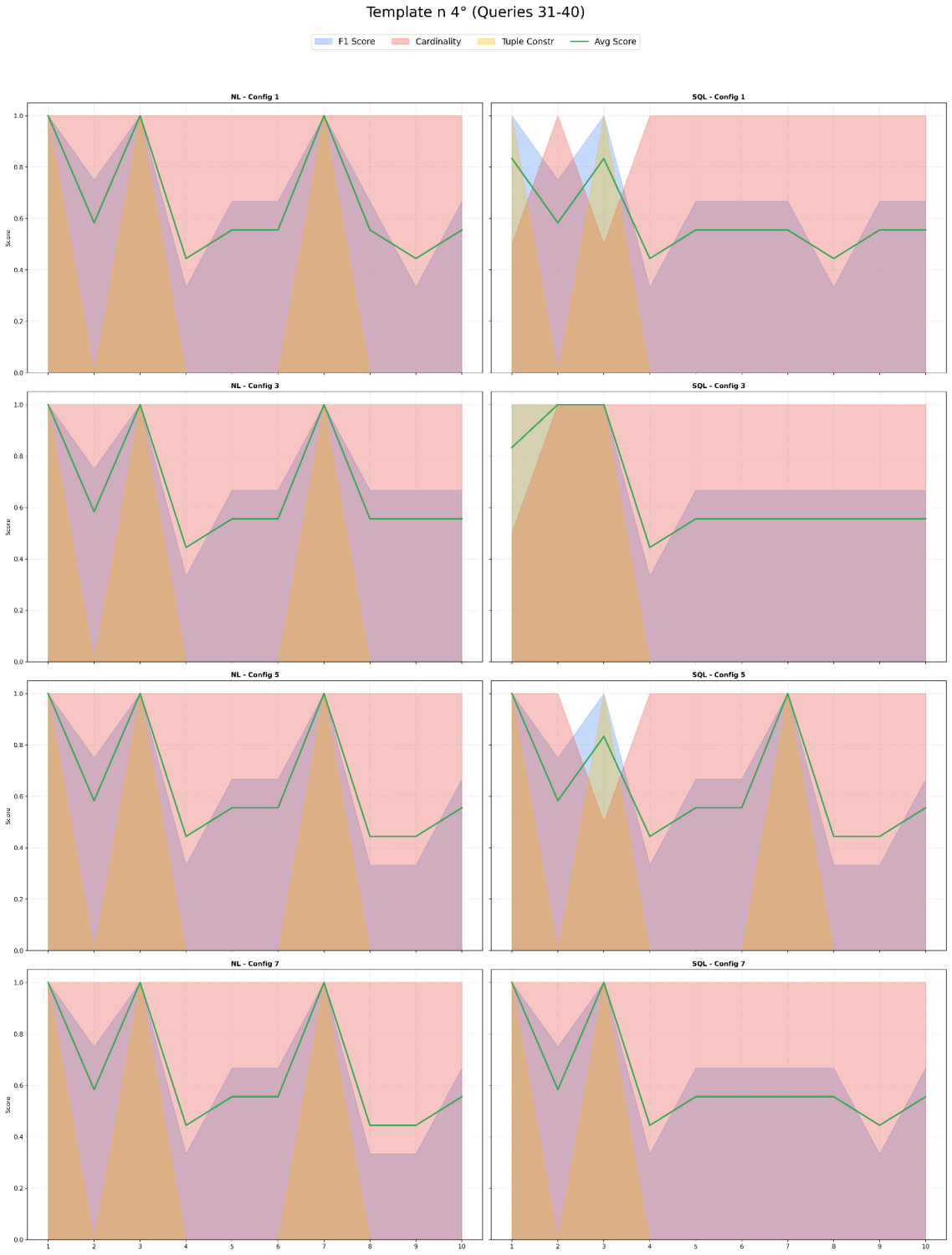
5.2.4 - Template number 4

In the fourth template, we ask GaloisPy to retrieve information about the award year and prize category again, similarly to what we did in the first template. However, this time, we will add an additional challenge by adding a JOIN operation, and asking to also provide the winner's country of birth. As before, the winner's name is specified in the WHERE clause.

```
SELECT a.award_year, a.category, w.birth_country
FROM nobel_prizes_1901_2023_awardees AS a
  JOIN nobel_prizes_1901_2023_recipients AS w
  ON a.name = w.name
WHERE a.name = 'Rita Levi-Montalcini';
```

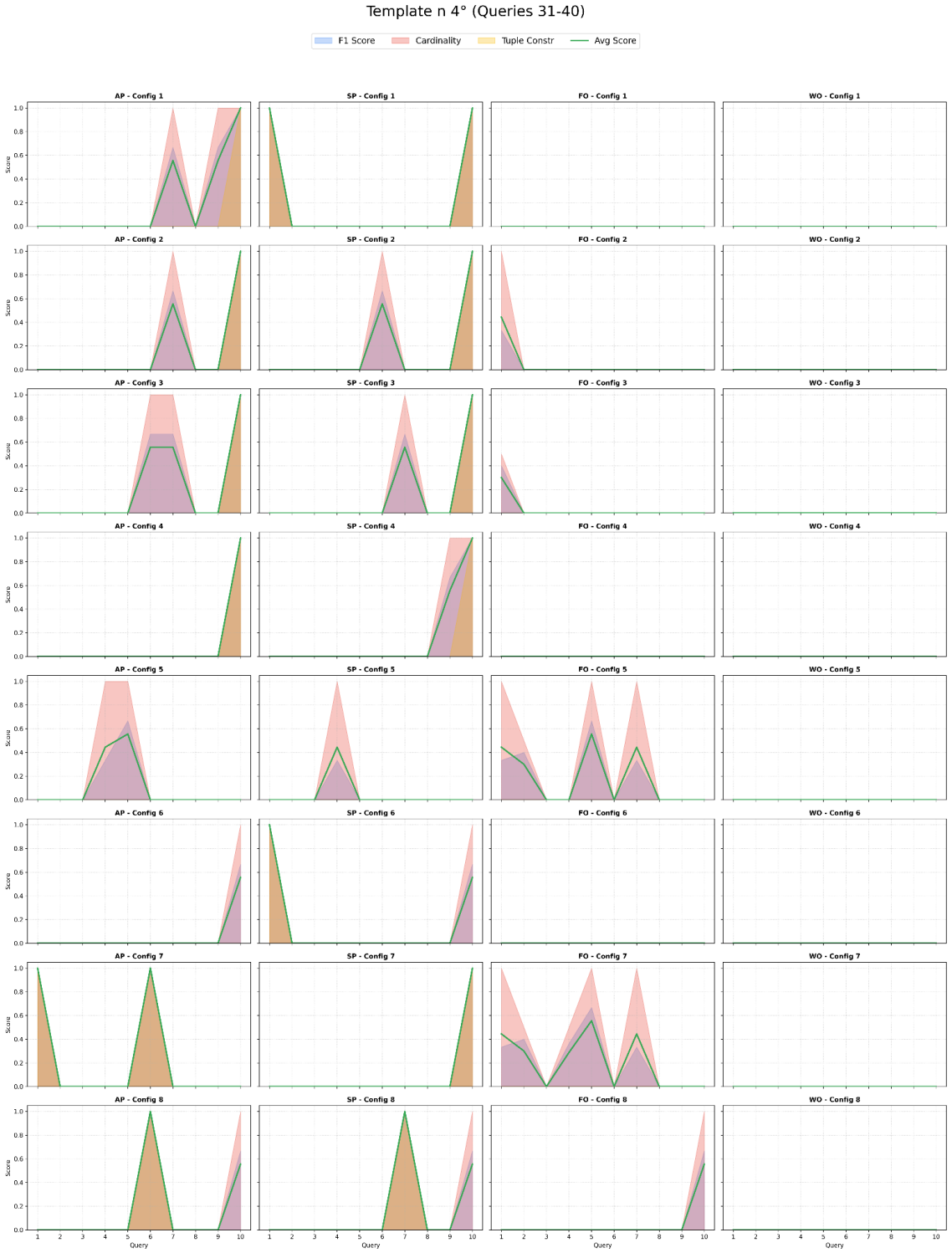
Figure 20 - Template 4 with JOIN clause and filtering condition over the winner name (in this example, Rita Levi-Montalcini).

Analyzing the performance achieved by the two baseline modes, we can see that both systems have worked pretty well. As we have seen before, in this fourth template, the best mode between the two is the NL one. This mode not only has always matched the expected cardinality, as highlighted by the Tuple Cardinality metric, but it has also achieved higher F1 Cell and Tuple Constraint values in almost all configurations tested. Anyhow, the SQL mode has also performed well, especially in configurations 3 and 7, highlighting the effectiveness of the retry strategy for this template.



Moving on to GaloisPy, also in this experiment, the various modes have answered sporadically, with AP and SP with similar performance, but overall poorly. Interestingly, the FO mode in some configurations, specifically configurations 5 and 7, has performed better than the other modes, answering around half of the queries. Last but not least, the

WO mode confirms once again all its limitations, producing once again no relevant result in this experiment.



In conclusion, from the data we got, we can state that the best modes were once again the two baselines, which answered to more queries in more configurations and with overall

higher performance. Different from what we have seen in experiment one, the GaloisPy system was not able to replicate the same performance seen previously, confirming our suspicions that the limitation resides in the management of the JOIN clause. We suspect that JOIN operations most of the time, due to the intersection operation between the two result sets, erase most of the tuples retrieved, leading to the poor performance we have just seen.

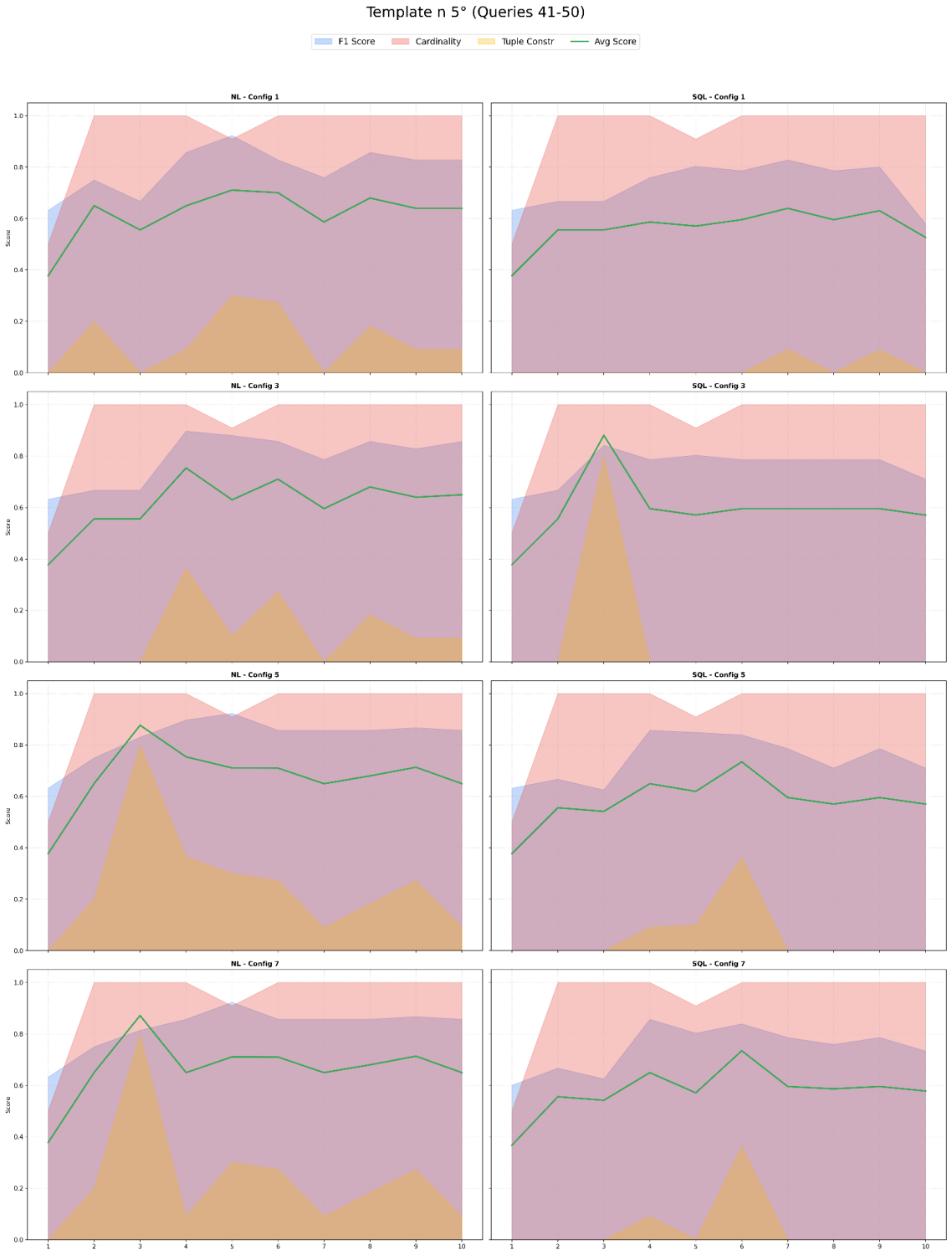
5.2.5 - Template number 5

The fifth and final template aims to analyze GaloisPy's ability to retrieve data within a specific time frame defined by the BETWEEN clause. In this template, we ask GaloisPy to provide the number of Nobel Prizes awarded each year within a specified time period.

```
SELECT award_year, COUNT(*) AS nobel_awarded
FROM nobel_prizes_1901_2023_awardees
WHERE award_year BETWEEN 1940 AND 1945
GROUP BY award_year;
```

Figure 21 - Template 5 with filtering condition over a specified temporal interval expressed by the BETWEEN clause (in this example, from 1940 to 1945).

As usual, we begin with the baseline modes. As the data below shows, both have performed well, with the NL mode performing slightly better than the SQL mode. Both have achieved high *Tuple Cardinality* and *F1 Cell* values in almost every query. Regarding the *Tuple Constraint*, however, the NL mode has been more accurate and has provided better results. Interestingly, both systems had difficulty with the first query, which was related to Nobel Prize awardees during World War II. This is probably because both systems returned data from 1940 to 1942, a period during which no Nobel Prizes were awarded due to the war.



Finishing with the GaloisPy's modes, from the data below, it clearly emerges that the best model among the tested was the AP. Its superior performance is probably because it pushes all the filtering conditions, which are, in this case, the beginning date and the ending date of the period of time, that focuses the LLM to retrieve data about such a

period. Nevertheless, the overall performance achieved is still a bit lower than the ones obtained by the baseline models.

In conclusion, we can state that, also in this experiment, the best models were the baseline ones, showing a superior performance in retrieving data of a specific temporal window.

On the other hand, only GaloisPy's AP mode has performed decently, suggesting that pushing down all conditions can help the model in retrieving the correct tuples. Anyhow, its performance has been lower than the baseline modes, showing some difficulties in such a template.

5.3 - Conclusions

From the experiments conducted, we can conclude that the two baseline models have performed significantly better than GaloisPy's ones, especially on queries with JOIN conditions. However, in queries that lack such an operator, GaloisPy's performance, especially for those who utilized the *Table-Scan* strategy, was comparable or even better, indicating the efficacy of our re-implementation.

Regarding GaloisPy's modes, from the experimental results, we can notice that the AP mode has generally performed better than the SP mode, demonstrating that providing the model with all filtering conditions aids in retrieving the correct values. On the other hand, the FO mode showed inconsistent performance, performing well on some queries but poorly on others, showing all its limitations. Lastly, the WO mode nearly produced no data in any of the experiments, highlighting the ineffectiveness of the *Key-Scan* mode without any optimizations, particularly for a small model like *GPT-4.1 nano*.

Conclusion and future work

The Galois system is one of the first software applications capable of retrieving structured data from LLMs using SQL syntax. Although it has achieved interesting results, it has several drawbacks, mostly related to its complex architecture. Therefore, we developed GaloisPy, a Python tool that overcomes most of these limitations by re-implementing all the core functionality of Galois in a simpler and more flexible way, thanks to a clearer and more extendable design. Additionally, GaloisPy introduces several improvements to enhance its customization and compatibility, such as various configuration files that users can modify and tailor to their needs, as well as features that improve its retrieval capabilities, like the prompt rewriting strategy or *Pydantic's BaseModels* that adapt dynamically to any query. Finally, GaloisPy can be easily imported and integrated into any existing projects thanks to its release as a *wheel* package and its class-based architecture, rather than a standalone application like Galois, making this tool easier to be integrated into many different scenarios.

Regarding the experiments we conducted, the results clearly show that, first of all, natural language prompts are more effective for LLMs. As demonstrated by the results, prompt rewriting significantly improved GaloisPy's performance, especially in the AP and SP modes. This indicates that providing the model with a prompt phrased as a natural language question, combined with *Pydantic's BaseModels*, can enhance the model's understanding of the task and thereby improve the overall quality of the generated results. Additionally, the experimental results reveal that GaloisPy, despite using more tokens, required fewer iterations than Galois, indicating a more efficient and clearer management of the generation process by the LLM, thanks to better handling of the chat memory history.

On the other hand, GaloisPy still presents some limitations that have undoubtedly limited its full potential. Starting from the management of JOIN conditions, we can see that, actually, this kind of operation is implemented in the same way as a traditional DBMS does, and, although this approach works perfectly for relational databases, this is not the best choice when it comes to LLMs. Indeed, from our tests' results, queries that have at

least one JOIN condition often return no results or at most a few of them because the intersection between the two result sets often leads to an empty set. A possible solution to this problem is to define a unified schema that includes both schemas involved in the JOIN clause into one, and then perform the scan operation over it. In fact, from our tests on the Nobel Prizes, we saw that in the first template, GaloisPy has performed pretty well, but when we changed the query structure by adding a JOIN operation, maintaining, however, almost the same request as in template four, most of the time, GaloisPy has not produced any result. Secondly, from the experiments we have done, we have noticed that all the GaloisPy's modalities that have used the *Key-Scan* strategy were the worst in terms of result quality, different from what was reported in the original paper. This suggests that querying the LLM model with simpler but more specific questions is counterproductive, and other scan strategies, such as the *Table-Scan*, provide solutions that are more effective and more efficient from all points of view. On the other hand, this inefficiency could also be related to the size of the LLM model we have used, but additional tests with bigger models are needed to confirm such a hypothesis.

In conclusion, GaloisPy certainly presents a valid re-implementation of the Galois system, outperforming it in almost any scenario. Indeed, thanks to the different enhancements developed, GaloisPy was able to achieve performance comparable or even superior to the ones obtained by Galois with larger LLM models like *GPT-4o mini*, *Llama 3.1 8b*, and, in rare cases, *Llama 3.1 70b*, even when using such a small model like the *GPT-4.1 nano*. However, its token usage and average time per query are slightly higher, but, as was previously presented in the experiments sections, it implements a more complex management of the chat memory that requires more tokens, but offers a clearer and more familiar structure, with higher usability. On the other hand, GaloisPy has inherited some limitations from the original system, such as the poor management of JOIN conditions and the ineffectiveness of the *Key-Scan* strategy, but overcoming these, this tool can be a valid starting point for the structured information retrieval field.

Finishing with the future of this implementation, at the moment, there are many alternative and valid solutions. We believe that one of the most interesting ones is to combine the RAG capabilities of GaloisPy with an external, always-updated source of information that provides authoritative documents to populate the vector database. For

example, it could be possible to use the free and open knowledge base, Wikidata, to obtain raw data on a specific topic, and then use the RAG GaloisPy system to execute the user's SQL queries. Another valid future implementation could be the evolution of the GaloisPy application into a new one, based maybe on AI agents. The integration of modern solutions, like the Model Context Protocol, or MCP, can bring the application to a whole new level of customization, in which different tools with different functionalities can be provided to the LLM model, opening the integration of several sources of knowledge from which GaloisPy can retrieve information, such as newspapers, wikis, or the whole web, throughout automation tools or browser engines that an AI agent can use.

Appendix A: Large language models

A.1 - LLMs and transformers

By the term large language models, or LLMs, we mean all the artificial intelligence models that are based on a deep neural network that is trained on a vast amount of textual data in self-supervised machine learning, a machine learning technique in which the machine uses the data itself to generate supervisory signals, rather than relying on externally-provided labels, that are designed for understanding and manipulating text. At the end of the training process, called pretraining, they are capable of understanding and working on natural language, or rather the language that humans use to communicate, and, as a result, they can handle a huge number of tasks such as translating a sentence from a certain language to another, also known as machine translation, answering questions or generating contents, also known as generative AI. Just to give an example of where this kind of technology is used, all modern chatbots like *ChatGPT*, *Gemini*, *Copilot*, and *Llama* are based on LLMs.

Now that we understand what LLMs are, let us dive more in-depth into the technology used to implement them and how the training process works. First, the technology. As was anticipated earlier, these models are based on deep neural networks. These networks are implemented using transformers, a neural network architecture developed by Google in 2017 that converts a sequence of text into another sequence of text. It is made of two components: an encoder and a decoder. The encoder converts the input text into a representation that condenses all the information into something smaller and more manageable for the machine to use and understand. The decoder, on the other hand, brings the information processed by the encoder and uses it to generate text, one token after the other, where a token can represent a whole word or a sub-word unit.

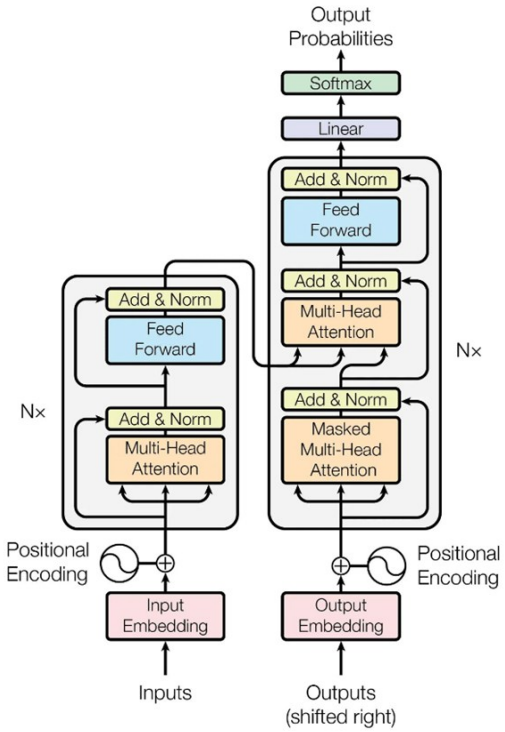


Figure 22 - Transformers architecture (on the left, the encoder, on the right, the decoder).

The peculiarity of this architecture is self-attention, a method that determines the importance of each component in a sequence relative to the other components in that sequence. It is like a sort of mind mapping that relates words together by their meaning, allowing artificial intelligence models to capture semantic information and relationships between different parts of the text. The self-attention works as follows: suppose we have the sentence *"Mark didn't go to school because he was sick"*. When the model receives this phrase as input, it processes one word after the other, and when it comes to the word *"he"*, it has to guess which word the word *"he"* is connected to. The self-attention mechanism weighs the semantic relation between the word *"he"* and all the previous words, and, as we expect, it will weigh more the relation *"Mark – he"*, creating a correlation between these two terms.

Moving on, let us now discuss the training process, named pretraining. During this phase, the neural network is trained over huge datasets, mainly text scraped from the web and filtered, in a self-supervised manner. At each time step t, we ask the model to predict the next word, and at the following step, we repeat the process, ignoring what the model has generated. This training mode is also known as *teacher forcing*.

As a result of pretraining, the language model acquires knowledge about the language and knowledge about the real world (notice that LLMs' knowledge entirely relies on the knowledge contained in the training corpora).

A.2 - Generation process

The generation process, or rather called inference, takes place after the training phase and consists of submitting a textual request to the model. When it receives the input from the user, the LLM model processes the input, and then it starts to produce a conditioned response one token after the other until it produces the so-called *End of Stream* (EOS) token, which indicates that the process is completed.

For this thesis, let us focus only on the process of choosing the next token to produce. As was anticipated before, the model produces one token after the other until it produces the EOS token, and the choice of which token to produce is statistical. Thus, the model chooses the next token to generate by random sampling, which is a probability-based technique used to select a sample from a larger group where every individual has a non-zero chance of being included, from the distribution over the vocabulary, which is the full list of tokens used by the machine. Therefore, tokens with higher probability are more likely to be chosen than tokens with low probability. To be clearer, consider the probability distribution like a dartboard and suppose that the LLM is the player: the larger the target area, the greater the likelihood of a dart piercing it.

However, the chance of not selecting the best option is not zero, so in case of applications in which reproducibility is crucial, as the project proposed in this thesis, a reshaping of the probability distribution is needed. To do so, there are two major techniques that improve but do not guarantee reproducibility, which are temperature and top-p.

Starting from the temperature, it is a parameter that ranges between 0 and 1, where low values smoothly increase the probability of the most probable words and decrease the probability of the rare words, while high temperature values flatten the word probability distribution.

$$P_i = \frac{e^{\frac{y_i}{T}}}{\sum_{k=1}^{n} e^{\frac{y_i}{T}}}$$

Equation 1 - Temperature formula.

Moving on to the top-p parameters, it does not modify the tokens' probability distribution, but it keeps the top p percent of the probability mass, thus it selects the smallest set of tokens whose total probability is greater than or equal to p.

$$\sum_{w \epsilon V^{(p)}} P(w|w_{<t}) \geq p$$

Equation 2 - Top-p formula.

Indeed, as these two parameters modify the probability distribution in different ways, they are often used jointly, and together they provide a high level of reproducibility, but, unfortunately, not absolute reproducibility.

Appendix B: Databases

B.1 - Relational databases

A relational database (RDB) is essentially a method of organizing information in table structures consisting of rows and columns. Each table represents an entity, or something meaningful to the user, such as a client, product, or order. This approach is highly flexible and customizable, enabling users to model and represent almost any kind of data in a simple, structured way.

After this brief introduction, let us dive into the structure of relational databases. First, a relational database is defined by a relational schema, which consists of one or more tables connected by relations, hence the name, and is established during the development phase. The schema describes the entities involved, which are the tables, the constraints that tables' attributes must follow, such as data types, allowed value ranges, and more, and the relationships between entities. Each row in a table represents a unique record, or tuple, composed of all the attributes defined in that table. To uniquely identify each record, a *primary key* must be designated, typically made up of one or more distinct attributes. Usually, integer values or codes are used as primary keys for simplicity. Primary keys are also used for entity relations. Indeed, a *foreign key* is a column, or an attribute, that refers to the primary key of another table. This mechanism allows the model to establish connections between different tables.

Component	**Definition**	**Function**
Table (*entity*)	A set of records organized in a table-like structure.	Represents an entity or a relationship between them.
Row (*record/tuple*)	Single element of an entity.	Contains data related to a single element of an entity.
Column (*attribute*)	A field that defines a particular aspect of the entity,	Defines the type and the nature of the data.
Schema	Logical structure of the database. It defines entities and relations among them.	Defines the overall logical structure and the entities involved.
Primary key	A set of attributes that uniquely identifies all the records in a table	Identifies records in a table.
Foreign key	Reference to an external primary key. Duplicates are allowed.	Establish connections between tables' data.

Table 9 - Summary table.

Before moving on, here is an example. Imagine you want to create a relational database to keep track of restaurant orders. First, we define the entities involved: tables and dishes. Next, we define the attributes of these two entities. For instance, a table should have a table number, a sector number, and the number of seats. Next, we specify the constraints for these attributes and identify a valid primary key, which in this case could be the combination of *"table number"* and *"sector"*. Finally, we define the relationships between the two tables. For example, we can specify an order relation involving one table and one or more dishes.

Finally, interaction with these structures occurs throughout the usage of the *Relational Database Management System* (RDBMS). These software tools supervise the archiving, retrieval, and safeguarding of data. Examples of RDBMS include systems such as MySQL, PostgreSQL, and DuckDB.

B.2 - SQL syntax and queries

SQL (Structured Query Language) is the standard tool to interact with relational databases. SQL implements a declarative language, or rather, the user specifies what data they want, and not how it should be retrieved, because it will be the database engine to define and optimize such an execution plan.

Before we begin to analyze the SQL syntax, let us focus on the mathematics. SQL queries are based on relational algebra[9], and its most important operators are:

- *Selection (σ):* It picks the records/tuples that satisfy a certain filtering condition. In SQL syntax, this operation corresponds to the *WHERE* clause.
- *Projection (π):* It selects a specific set of attributes, ignoring the others. This operation is defined through the *SELECT* SQL statement.

[9] It is a procedural query language, developed by Edgar F. Codd in 1970, that serves as the theoretical foundation for relational databases and SQL. It operates on relations (tables) using operators like selection, projection, and join to produce new relations, making it essential for query optimization and database design.

- *Cartesian product (×):* It combines all the rows of a table with all the rows of another table. In SQL syntax, this operation is performed by the *CROSS JOIN* operator.
- *Join (⋈):* It is a combination of the Cartesian product and selection based on an equality condition. This corresponds to the *JOIN* operation.

Moving forward, let us understand the structure of SQL queries. For the purposes of this thesis, we will present a simple yet flexible SQL structure that can be used in many different scenarios, but please note that several other operators are available. The syntax is reported in the figure below.

query.sql

```
SELECT [DISTINCT]? [attrs]
FROM [tab]
[LEFT | RIGHT | INNER | OUTER]? JOIN [tab] ON [cond]
WHERE [conds]
GROUP BY [attrs]
HAVING [conds]
ORDER BY [attrs] ASC | DESC
LIMIT [numb];
```

Figure 23 - SQL syntax.

Let us now analyze the various components of such syntax. The operators are:

- *SELECT:* It is a mandatory clause and indicates which attributes should be present in the result set. It is also possible to use operations over attributes, often referred to as aggregation functions like SUM, AVG, COUNT, and others, and the DISTINCT clause, which removes duplicate values for such attributes.
- *FROM:* This clause is also mandatory, and it indicates the table from which data should be extracted.
- *JOIN:* It is optional, and it is used to perform join operations. There are several types of join:
    - *LEFT:* It returns all records from the left table, and the matched records from the right table.
    - *RIGHT:* It returns all records from the right table, and the matched records from the left table.

    - *INNER:* It returns records that have matching values in both tables
    - *OUTER:* It returns all records when there is a match in either the left or the right table.

  If no JOIN type is expressed, it will be used as the INNER one. Therefore, the join condition is expressed by the ON operator, which specifies the two attributes, one from each table, involved in the join condition.
- *WHERE:* It is used to express filtering conditions among attributes. Multiple conditions can be concatenated throughout the logical operators AND and OR. This clause is optional.
- *GROUP BY:* It is a statement that aggregates rows that have the same values into a single row. This clause is often used with aggregation functions.
- *HAVING:* This clause is similar to the WHERE condition, but it is used to apply filtering conditions over aggregation functions. It can be used if and only if a GROUP BY clause is used.
- *ORDER BY:* It orders the result set according to the value of the attributes involved in this operator in ascending order (ASC) or in descending order (DESC). If no ordering clause is expressed, tuples will be ordered in ascending order.
- *LIMIT:* It is used to limit the results returned from the RDBMS. The value expressed from the LIMIT clause is the maximum number of tuples that can be returned.

Before we conclude, here is an example. Imagine you want to retrieve all the items listed in all your clients' pending orders, ordered by date in descending order. Supposing that item information is stored in *item_id* and *item_descr*, a possible query can be the following.

pending-orders.sql

```sql
SELECT item_id, item_descr
FROM client_orders
WHERE status = 'pending'
ORDER BY issue_date DESC;
```

Figure 24 - Pending orders query.

Bibliography and webography

Books and articles:

[1]. Atzeni, P., Ceri, S., Paraboschi, S., & Torlone, R. (1999). *Database Systems - concepts, languages & architectures.*

[2]. Di Nunzio, G.M., & Di Buccio, E. (2017). *Basi di dati: progettazione concettuale, logica e SQL.*

[3]. Jurafsky, D., & Martin, J. H. (2025). *Speech and Language Processing.*

[4]. Saeed, M., De Cao, N., & Papotti, P. (2024). *Querying Large Language Models with SQL.*

[5]. Satriani, D., Veltri, E., Santoro, D., Rosato, S., Varriale, S., & Papotti, P. (2025). *Logical and Physical Optimizations for SQL Query Execution over Large Language Models.*

[6]. Sommerville, I. (2016). *Software Engineering.*

[7]. Yu, T. et. al. (2019). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task.*

Webography and software resources:

[1]. Langchain, Official Repository and Documentation, GitHub – *https://github.com/langchain-ai/langchain*

[2]. Llama-index, Official Repository and Documentation, GitHub – *https://github.com/run-llama/llama_index*

[3]. Mixedbread, mxbai-embed-large-v1, Hugging Face – *https://huggingface.co/mixedbread-ai/mxbai-embed-large-v1*

[4]. Mixedbread, mxbai-rerank-xsmall-v1, Hugging Face – *https://huggingface.co/mixedbread-ai/mxbai-rerank-xsmall-v1*

[5]. Ollama, Official Website – *https://ollama.com/*

[6]. OpenAI, tiktoken, GitHub – *https://github.com/openai/tiktoken*

[7]. University of Basilicata and EURECOM, Galois official release, GitHub – *https://github.com/dbunibas/galois*

[8]. Zaki, A. M., Nobel Prize Winners Dataset (1901-2025), Kaggle – *https://www.kaggle.com/datasets/ahmeduzaki/nobel-prize-winners-dataset-1901-2025*