Enabling Structured Query Execution over Disorganized Data Lakes

Strukturierte Abfrageausführung über unstrukturierten Data Lakes
Bachelor thesis by Jakob Steinke
Date of submission: November 30, 2025

1. Review: Professor Dr. Carsten Binnig
2. Review: Jan-Micha Bodensohn, M.Sc.
Darmstadt

Computer Science Department
Data and AI Systems

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Jakob Steinke, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 30. November 2025

Jakob Steinke

Abstract

Modern organizations increasingly rely on data lakes to store vast amounts of heterogeneous and loosely structured data. While this flexibility simplifies large-scale data integration, it complicates querying the data: unlike traditional relational databases, data lakes lack schema information, including explicit table and column names and foreign-key relationships. Additionally, they often contain noisy or inconsistent values, ambiguous naming conventions, and partially overlapping tables. As a result, querying the data lake involves making assumptions about the underlying structure and content, which can lead to various errors.

This thesis addresses the challenge of enabling structured query execution over disorganized data lakes through an iterative SQL rewriting framework powered by Large Language Models (LLMs). The proposed system iteratively transforms user-written SQL queries into executable queries that operate correctly on the actual data lake and fulfill the user's intent. To achieve this, the framework integrates an iterative feedback loop in which the system repeatedly analyzes execution results and performs targeted retrieval and transformation steps, such as searching for missing tables, discovering join paths, or cleaning noisy data, to refine the query and align it with the data lake's structure.

An extensive evaluation over a curated benchmark of 103 query–result pairs derived from the WikiDBs corpus demonstrates that iterative rewriting substantially improves query execution accuracy compared to strong single-step baselines. Ablation studies further reveal which components of the system contribute most strongly to performance gains.

Overall, this work presents the first systematic evaluation of LLM-driven query rewriting over noisy, schema-free data lakes. The findings highlight the potential of iterative reasoning to bridge the gap between user intent and the heterogeneity of real-world data, paving the way for more robust, adaptive data lake querying systems.

Zusammenfassung

Moderne Unternehmen verlassen sich zunehmend auf Data Lakes, um große Mengen heterogener und lose strukturierter Daten zu speichern. Diese Flexibilität vereinfacht zwar die Integration großer Datenmengen, erschwert jedoch die Abfrage der Daten: Im Gegensatz zu herkömmlichen relationalen Datenbanken fehlen Data Lakes Schemainformationen wie explizite Tabellen- und Spaltennamen sowie Fremdschlüsselbeziehungen. Darüber hinaus enthalten sie oft verrauschte oder inkonsistente Werte, mehrdeutige Namenskonventionen und teilweise überschneidende Tabellen. Infolgedessen erfordert die Abfrage von Data Lakes Annahmen über die zugrunde liegende Struktur und den Inhalt, was zu einer Vielzahl unterschiedlicher Fehler führen kann.

Diese Arbeit befasst sich mit der Herausforderung, die strukturierte Ausführung von Abfragen über unorganisierte Data Lakes mithilfe eines iterativen SQL-Umschreibungsframeworks zu ermöglichen, das auf Large Language Models (LLMs) basiert. Das vorgeschlagene System wandelt vom Benutzer geschriebene SQL-Abfragen schrittweise in ausführbare Abfragen um, die auf dem tatsächlichen Data Lake korrekt funktionieren und die Absicht des Benutzers erfüllen. Um dies zu erreichen, integriert das Framework einen iterativen Feedback-Loop, in dem das System die Ausführungsergebnisse wiederholt analysiert und gezielte Retrieval- und Transformationsschritte durchführt, wie z. B. die Suche nach fehlenden Tabellen, das Auffinden von Join-Pfaden oder die Bereinigung von verrauschten Daten, um die Abfrage schrittweise zu verfeinern und an die Struktur des Data Lakes anzupassen.

Eine umfassende Auswertung anhand einer eigens erstellten Benchmark von 103 Abfrage-Ergebnis-Paaren aus dem WikiDBs-Korpus zeigt, dass iteratives Umschreiben die Genauigkeit der Abfrageausführung im Vergleich zu starken einstufigen Baselines erheblich verbessert. Ablationsstudien zeigen darüber hinaus, welche Komponenten des Systems am stärksten zur Leistungssteigerung beitragen.

Insgesamt stellt diese Arbeit die erste systematische Evaluation LLM-gesteuerter Abfrageumschreibung über verrauschte, schemalose Data Lakes dar. Die Ergebnisse verdeutlichen das Potenzial iterativer Reasoning-Ansätze, um die Diskrepanz zwischen der Abfrageabsicht des Nutzers und der Heterogenität realer Datenquellen zu überbrücken, und ebnen den Weg für robustere und anpassungsfähigere Data-Lake-Abfragesysteme.

Contents

1 Introduction

1.1 Motivation

Modern organizations increasingly rely on *data lakes* to store large amounts of heterogeneous data collected from various sources. Compared to *relational databases*, data lakes do not enforce a well-defined schema and contain loosely structured or even disorganized tables that originate from independent data collection processes. Although this flexibility enables scalable data integration, it also introduces significant challenges for querying and analysis. In traditional relational databases, users can formulate precise SQL queries that operate on a known schema with clearly defined foreign-key relationships and consistent data types. In contrast, querying a data lake is far more complex. The schema is typically unknown, and tables frequently exhibit overlapping semantics, incomplete or missing relationships, and inconsistent naming conventions. For instance, a user may issue a simple query such as:

```sql
SELECT country.name, SUM(site.revenue) AS revenue
FROM country
JOIN site ON country.id = site.cid
WHERE country.population > 1000000
GROUP BY country.name
ORDER BY revenue;
```

Listing 1: Example SQL query assuming a clean relational schema.

However, this query may fail on a data lake because there is no table named `site`, no column exactly called `revenue`, and no explicit foreign-key relationship that specifies how these tables should be connected. In addition, the underlying data may contain noise, such as irregular text encodings or inconsistent categorical values, further hindering correct execution.

To answer the query, the user would have to search for relevant tables manually, determine how to connect them, look up specific values, and clean up noisy entries. This process is both time-consuming and error-prone, often requiring substantial domain knowledge and repeated exploration before obtaining a correct result.

Despite these challenges, the ability to make data lakes as easy to query as relational databases is increasingly valuable. Large-scale industry datasets, open government data portals, and multi-domain knowledge integration projects all depend on extracting structured insights from messy, semi-structured repositories. Automating this process would drastically reduce the manual effort required for schema exploration, data cleaning, and query adaptation, tasks that still require expert knowledge of both the schema and its inconsistencies.

To address this gap, recent research explores how Large Language Models (LLMs) can assist users in querying data. For example, Text-to-SQL systems translate natural-language questions into executable SQL queries, enabling intuitive database access. However, most existing approaches assume clean and well-structured schemas with reliable table relationships. On the other hand, Open Table Question Answering derives insights from disorganized tables but is limited to simple queries, lacking the power and rigor of SQL.

This motivates our work: we aim to enable users to execute SQL queries over data lakes by rewriting them to operate on the actual structure and content of the available tables.

In the following section, we outline this challenge more formally and present the research questions guiding this thesis.

1.2 Problem Statement

The goal of this thesis is to make querying data lakes as intuitive and reliable as querying traditional relational databases. Formally, the input to our system is a user-defined SQL query Q_{user} together with a data lake $L = \{T_1, T_2, \ldots, T_n\}$, which contains a large collection of independent, heterogeneous tables. The user formulates Q_{user} under the assumption of a coherent schema that fits their query. The desired output is a result table R_{out} that correctly reflects the intent of the user query when executed over the relevant tables within the data lake.

Figure 1.1 illustrates this setting. Given an arbitrary user query that assumes a coherent database schema, the system must first identify relevant tables, align schema mismatches, and construct a valid rewritten query that can be executed on the disorganized data lake.

Motivating Example. Consider again the example query shown in Listing 1. When executed on a relational database, this query succeeds because it was written to match the known schema. In a data lake, however, no such knowledge exists: tables are only loosely related, may use inconsistent or ambiguous names, and often lack explicit foreign-key relationships.

To answer the query correctly, the system must automatically:

- retrieve the required tables from the data lake,
- infer possible join paths between them,
- rewrite the query so that it runs on the retrieved tables,
- clean or normalize noisy columns when necessary.

The core idea of this approach is to iteratively rewrite the user query using LLMs. Starting from the initial query that assumes a coherent schema, the system repeatedly retrieves relevant tables, proposes rewritten queries, executes them, and evaluates the results. Through this feedback loop, the query is iteratively refined until it becomes executable on the actual data lake.

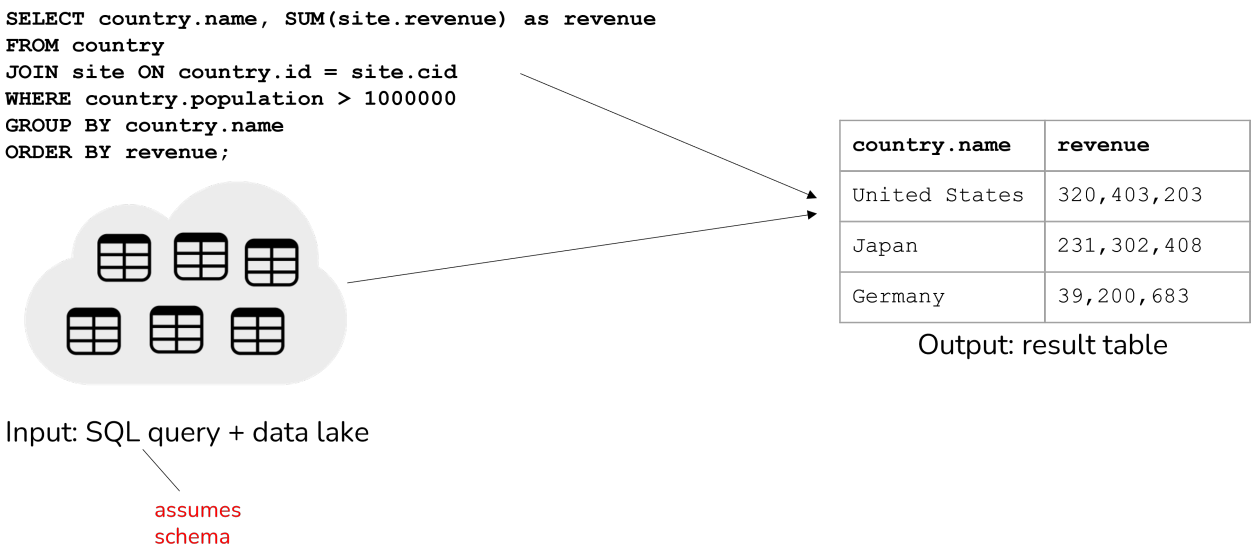

Figure 1.1: **Overview of the problem setting**. The figure illustrates the central challenge of this work: a user formulates an SQL query under the assumption of a coherent schema, while the data lake contains many loosely structured and disconnected tables. The system must resolve this mismatch to execute the query as intended.

Challenges. This problem involves multiple challenges: First, we need to retrieve the subset of tables relevant to the user query from a large, heterogeneous data lake. Next, we must handle schema mismatches by resolving incorrect or inconsistent column and table names and compensating for missing foreign-key relationships. We then need to perform join reasoning to discover plausible join paths between tables in the absence of explicit schema information. Finally, we must address data noise by detecting and correcting corrupted or irregular values to ensure robust query execution. In building our system and addressing these challenges, we want to answer the following research questions:

Research Questions.

1. **RQ1:** How can we iteratively transform SQL queries so that they operate correctly over large, heterogeneous, and schema-free data lakes?
2. **RQ2:** How can we enable the translation agent to interact with the data lake to find relevant tables, infer join paths, and adapt the query to the underlying structure and content?
3. **RQ3:** How can the system detect and mitigate noisy or inconsistent data to ensure reliable and accurate query execution?

Together, these research questions guide the design and evaluation of our proposed system.

1.3 Goals and Contributions

The overall goal of this thesis is to develop and evaluate a system that enables users to execute SQL queries over heterogeneous data lakes as seamlessly as over traditional relational databases. To achieve this, the work primarily focuses on the algorithmic design of an iterative query rewriting framework powered by LLMs. In addition, it introduces a reproducible evaluation framework to systematically assess the system's performance under common data lake challenges such as schema mismatch, missing joins, and noisy data.

Goals. Building on the problem formulation introduced above, this thesis pursues three main objectives:

- Develop an iterative SQL rewriting framework that progressively transforms an initial user query into an executable version by leveraging LLMs for table retrieval, join reasoning, and data normalization.
- Enable effective interaction with the data lake by designing retrieval and reasoning mechanisms that identify relevant tables, infer join paths, and adapt queries to the data lake's heterogeneous structure and content.
- Establish a reproducible evaluation framework to systematically benchmark query rewriting performance and accuracy under realistic data lake challenges, including schema mismatch, missing joins, and noisy data.

Contributions. To accomplish these goals, this thesis makes the following concrete contributions:

1. **Evaluation Dataset from WikiDBs.** We construct a challenging benchmark dataset derived from the WikiDBs corpus (Vogel et al. 2024). It consists of 103 user queries designed with controlled schema mismatches, missing joins, and noisy data, each paired with a verified gold-standard SQL query and result table for quantitative evaluation.
2. **Retriever–Reader and Retriever–Rewriter Approaches.** We compare two strong baseline approaches for SQL execution over data lakes: a Retriever–Reader model that directly predicts the result table from retrieved tables, and a Retriever–Rewriter model that generates an executable SQL query in a single step.
3. **Iterative SQL Rewriting Framework.** We design and evaluate the proposed Iterative Rewriter architecture, in which multiple LLM agents iteratively refine the query through self-checking, join path discovery, and adaptive noise handling via dynamically generated UDFs.
4. **Comparative Evaluation of LLMs.** We conduct an empirical comparison of different language models, namely GPT-4.1 mini, GPT-5 mini, and GPT-5 nano, in terms of query accuracy, cost, and latency.

Together, these contributions provide the first end-to-end evaluation of LLM-driven SQL rewriting over disorganized data lakes, establishing a foundation for future research on adaptive data exploration and automated schema reasoning.

1.4 Thesis Outline

The rest of this thesis is structured as follows: Chapter 2 provides the theoretical background, contrasting databases and data lakes, and introduces related research directions in Text-to-SQL, Open Table Question Answering, and join-aware retrieval. Chapter 3 presents the overall system architecture, explaining the iterative rewriting loop, its main components, and the specialized data lake actions that guide retrieval and transformation. Chapter 4 focuses on the iterative rewriting process itself, detailing the rewriter and self-checker prompts, the feedback loop, and the management of the scratchpad across iterations. Chapter 5 explains how relevant tables are identified through Query–Table and Table–Table Relevance, and how joinability is estimated using the precomputed Join Graph. Chapter 6 introduces the noise-handling component, showing how the system detects and cleans inconsistent data using LLM-defined User Defined Functions (UDFs) and Common Table Expressions (CTEs). Chapter 7 describes the evaluation setup, including dataset design, baselines, and experimental results, followed by ablation and behavioral analyses. Chapter 8 discusses the implications and limitations of the findings, and Chapter 9 concludes the thesis with a summary and outlook on future research directions for LLM-driven SQL rewriting over data lakes.

2 Background

This section provides the conceptual and technical background necessary to understand the problem setting of this thesis. We first contrast traditional relational databases with data lakes to highlight the specific challenges that arise in our setting. We then review related research areas that address similar goals, including Text-to-SQL and Open Table Question Answering (OTQA). Finally, we discuss four related frameworks from the literature: TAG, MMQA, Join-Aware Retrieval, and MURRE, which inspire different components of our proposed approach.

2.1 Databases vs. Data Lakes

Traditional relational databases are designed around a well-defined and normalized schema. Each table represents a specific entity or relationship, and explicit foreign-key constraints encode how tables can be joined. This design enables users to issue complex SQL queries that reliably produce correct results as long as the schema is known. The schema acts as a contract between the data and the query logic: column names, types, and relationships are clearly specified, enabling static query validation and predictable execution.

In contrast, data lakes are loosely structured repositories that aggregate data from multiple, often heterogeneous, sources. They typically store tables extracted from web data, spreadsheets, APIs, or domain-specific databases without enforcing a consistent schema or naming convention (Hai et al. 2021). Consequently, data lakes are flexible but noisy: tables may overlap semantically, have inconsistent or missing keys, and contain irregular data formats. Even seemingly simple queries can be challenging to execute, since the required tables must first be discovered, matched, and connected via inferred relationships.

While relational databases support declarative querying over clean, typed data, querying data lakes requires additional reasoning about retrieval, schema alignment, and noise handling. This thesis builds on this distinction and treats data lakes as large, disorganized collections of tables, where the system must map the user query to the actual tables of the data lake before it can be executed.

2.2 Related Tasks: Text-to-SQL and Open Table Question Answering

The problem of interpreting a user query and executing it on structured data has been studied in various forms. Two related tasks are particularly relevant to our setting: Text-to-SQL and OTQA.

Text-to-SQL. In the classical Text-to-SQL problem, the input is a natural-language question and a relational database. The output is a syntactically correct SQL query that answers the question. Modern approaches use LLMs to translate natural-language inputs into executable SQL queries and are typically evaluated on benchmarks such as Spider Yu et al. (2019), BIRD Li et al. (2023), and Spider 2.0 Lei et al. (2025). However, these models assume the database schema is known and consistent. They do not perform table retrieval or schema matching, since the relevant tables are already defined. This makes Text-to-SQL an idealized setting compared to querying real data lakes.

Open Table Question Answering (OTQA). OTQA generalizes Text-to-SQL by removing the assumption of a fixed schema. Instead, the system must first retrieve relevant tables from a large corpus before reasoning over them. The input is a natural-language question and a large collection of tables, while the output is a textual answer rather than a SQL query (Herzig et al. 2021). Systems in this area typically operate on self-contained tables with minimal relational structure, whereas our setting requires reasoning over multi-table joins and schema mismatches.

Comparison. Table 2.1 summarizes the distinction between these tasks and the setting addressed in this thesis. Our problem differs from both Text-to-SQL and OTQA by requiring the execution of a given SQL query over a disorganized data lake. The system must thus combine retrieval, schema alignment, and iterative query rewriting to bridge the gap between the user's assumptions and the true structure of the data.

Table 2.1: Comparison of related problem settings.

Aspect	Text-to-SQL	OTQA	Our Problem Setting
Input	Natural language question + relational database	Natural language question + data lake	SQL query + data lake
Output	SQL query	Natural language answer	Result table
Retrieval	None (fixed schema)	Top-k table retrieval	Iterative top-k table retrieval
Data structure	Clean relational schema	Self-contained tables, no joins	Noisy tables, missing joins, schema mismatch

2.3 Related Approaches and Research Directions

In this section, we present related approaches and explain how they motivate the design and research directions of our proposed system.

TAG: Text2SQL + Retrieval-Augmented Generation. The TAG framework (Biswal et al. 2024) integrates classical Text-to-SQL generation with retrieval-augmented generation (RAG) over clean databases. TAG demonstrates that combining symbolic SQL planning with contextual retrieval can improve query generation accuracy when partial schema information is available. However, TAG assumes well-defined relational schemas without schema mismatch or noisy joins. Our approach extends this idea to disorganized data lakes by introducing iterative query rewriting and retrieval feedback, rather than one-shot SQL generation.

MMQA: Multi-Table Multi-Hop Reasoning. MMQA (Wu et al. 2024) evaluates LLMs on multi-hop question answering across multiple tables. While its task formulation is closer to open-domain reasoning, it focuses on producing natural-language answers rather than executable SQL or result tables. Its core idea is to decompose complex reasoning into sequences of table-level operations, which motivates our iterative design: each iteration corresponds to a reasoning hop that extends the current query with additional joins or normalization steps.

Join-Aware Retrieval. Chen et al. (2025) propose a join-aware multi-table retrieval model that explicitly considers table–table compatibility when ranking candidate tables. Instead of retrieving tables independently, their retriever computes joinability scores to identify sets of tables that can be combined to answer a query. We adopt this idea by precomputing a *Join Graph* over the data lake, which captures schema- and instance-level similarity between tables and allows efficient discovery of plausible join paths during rewriting.

MURRE: Iterative Multi-Hop Retrieval. Finally, the MURRE framework (Zhang et al. 2024) introduces iterative retrieval with removal, where previously used or low-relevance tables are discarded in subsequent iterations. This idea parallels the feedback-driven retrieval loop in our system: after each rewriting step, we evaluate execution results and selectively expand or prune the context based on retrieved evidence. MURRE was originally designed for open-domain Text-to-SQL, but its iterative retrieval and self-correction mechanisms align closely with our iterative rewriting strategy.

Taken together, these approaches reveal complementary insights that inform the design of our approach:

- TAG motivates integrating retrieval with symbolic SQL reasoning.
- MMQA motivates multi-hop decomposition of complex queries.
- Join-Aware Retrieval motivates joinability-based ranking of candidate tables.
- MURRE motivates iterative retrieval and self-correction to refine context over time.

Our system combines these ideas to form a unified, iterative rewriting pipeline capable of executing SQL queries over schema-less, noisy data lakes.

2.4 Dataset Foundation

The WikiDBs corpus (Vogel et al. 2024) serves as the foundation of our evaluation. WikiDBs is a large-scale open-source dataset derived from Wikidata, containing approximately 100,000 relational databases across diverse domains, including people, countries, organizations, and cultural artifacts. Each database consists of multiple interconnected tables that resemble realistic relational structures, including foreign-key relationships and both textual and numerical attributes. Unlike purely synthetic benchmarks, WikiDBs is grounded in real-world knowledge and captures many of the irregularities and inconsistencies typical of human-curated data.

For this work, we select a representative subset of around 100 tables from one WikiDBs database to serve as our data lake. This configuration offers a realistic yet controlled environment for studying SQL query rewriting over heterogeneous data: Tables overlap semantically, vary in naming conventions, and feature missing or inconsistent links between tables. Moreover, while the original dataset includes schema metadata, we intentionally omit foreign-key information to simulate the schema-free nature of real data lakes. This setup enables rigorous testing of our system's capabilities in table retrieval, join reasoning, and query rewriting under challenging real-world conditions.

3 Architecture Overview

The input of our system is a user query that likely makes wrong assumptions about the underlying data of the data lake. First, we retrieve potentially relevant tables for the query. Next, the *rewriter* rewrites the query based on the provided tables. The rewritten query is then executed and forwarded to the *self-checker*, which examines the result and decides the next actions to take. This entire process is repeated until the self-checker chooses to return a result. Intermediate results are stored in a *rewrite history* to avoid repetition. Relevant data is stored in a *scratchpad* to use in subsequent iterations. Figure 4.1 gives an overview of this approach. In the following section, we motivate this high-level approach using an example query. In later sections, we will delve deeper into the individual components and provide further details.

3.1 Example Use Case

Suppose the user wants to execute the following query:

```sql
SELECT
    name,
    is_un_member,
    gini_index
FROM
    countries
WHERE
    currency = 'USD'
ORDER BY
    name;
```

Initial Retrieval First of all, we retrieve tables that are semantically similar to the schema assumed by the user. Thus, we search for tables about countries with columns `name`, `is_un_member`, `gini_index` and `currency`. This initial retrieval step reveals to us a table called `country_profiles` with the columns `country_name`, `in_un`, and `currency`.

```
Retrieved tables (iteration 1)
- country_profiles (country_name, in_un, currency, iso3, ...)
- languages        (language, country_iso3, region, ...)
- football_clubs   (club_name, country, stadium, ...)
...
```

The rewriter uses this information to create the following query:

```
SELECT
    country_name AS name,
    in_un        AS is_un_member,
    gini_index
FROM
    country_profiles
WHERE
    currency = 'USD'
ORDER BY
    country_name;
```

However, executing this query shows us that we incorrectly assume that this table also contains a column `gini_index`.

Execution Error

```
no such column: gini_index
```

We add this output and the rewritten query to the rewrite history to prevent making the same mistake in upcoming iterations.

Expanding the Schema The self-checker reacts to the execution error by explicitly searching for tables that contain a column similar to `gini_index`. This search yields the table `inequality_stats` with the column `gini_coef`. A preview of the table is added to the scratchpad, providing context for both the rewriter and the self-checker in the following iterations.

Finding Join Paths The next question is how to join the tables `country_profiles` and `inequality_stats`. To do so, the self-checker searches for a way to join both tables and detects the direct join via `country_profiles.iso3 = inequality_stats.iso3`.

Fixing the Filter Predicate Next, we have to verify the join predicate `WHERE currency = 'USD'`. For this task, the self-checker orchestrates a search for `'USD'` in `country_profiles` and finds the value `'US Dollar'` in the `currency` column, which the rewriter can use to adjust the predicate.

Scratchpad

```
Table preview: country_profiles
-------------------------------------------------------------
| country_name    | is_un | iso3 | currency            | gini |
|-----------------|-------|------|---------------------|------|
| United States   | t     | USA  | US Dollar           | 0.414|
| Ecuador         | true  | ECU  | US Dollar           | 0.456|
| "Taiwan"        | 0     | TWN  | New Taiwan Dollar   | 0.342|
-------------------------------------------------------------
...
```

Handling Noise Looking at the table previews in the scratchpad, we can tell that the columns `country_name` and `is_un` seem to suffer from noisy data. Consequently, the self-checker decides to define custom UDFs to clean the data and to store a preview of the cleaned table as a CTE in the scratchpad. Because we let the LLM define custom UDFs rather than relying on predefined ones, our system can adapt to various types of noise. For example, `parse_bool` can reliably transform values like `trüe` or `t`.

Scratchpad

```
...

CTE previews:

clean_countries AS (
  SELECT *,
         normalize_text(country_name) AS country_name_norm,
         parse_bool(is_un)            AS is_un_member_norm,
         normalize_text(currency)     AS currency_norm
  FROM country_profiles
)

Preview:
-------------------------------------------------------------------
| country_name_norm | is_un_member_norm | iso3 | currency_norm     | ...
|-------------------|-------------------|------|-------------------|-----
| United States     | true              | USA  | US Dollar         | ...
| Ecuador           | true              | ECU  | US Dollar         | ...
| Taiwan            | false             | TWN  | New Taiwan Dollar | ...
-------------------------------------------------------------------
...
```

With this new evidence, the rewriter produces the following query:

```sql
WITH clean_countries AS (
  SELECT *,
         normalize_text(country_name) AS country_name_norm,
         parse_bool(is_un)            AS is_un_member_norm
  FROM country_profiles
)
SELECT
  c.country_name_norm AS name,
  c.is_un_member_norm AS is_un_member,
  is2.gini_coef       AS gini_index
FROM clean_countries AS c
JOIN inequality_stats AS is2
  ON c.iso3 = is2.iso3
WHERE c.currency = 'US Dollar'
ORDER BY c.country_name_norm;
```

Executing it yields the following result:

```
Execution Result (Preview, 3 rows)
-------------------------------------------------------------
| country_name   | is_un | iso3 | currency           | ... |
|----------------|-------|------|--------------------|-----|
| Ecuador        | true  | ECU  | US Dollar          | ... |
| United States  | true  | USA  | US Dollar          | ... |
| Taiwan         | false | TWN  | New Taiwan Dollar  | ... |
-------------------------------------------------------------
```

Termination Finally, the self-checker assesses the new execution outcome to look sufficiently accurate and outputs the rewritten query.

3.2 Main Components

The architecture is structured around six main components:

- **Retriever:** In each iteration, we retrieve tables from the data lake that might be relevant to rewrite the query. Similar to related work, we include both direct similarity to the user query (Query-Table Relevance) and compatibility with already retrieved tables (Table-Table Relevance). Query-Table Relevance is computed once at the start of the approach, whereas Table-Table Relevance is recomputed every iteration based on the tables stored in the scratchpad.
- **Rewriter:** An LLM agent responsible for generating a rewritten version of the user query that is executable on the data lake. It is prompted to propose a new query that extends or refines the search space, for example, by joining additional tables or normalizing column references.
- **Self-Checker:** A second LLM agent that evaluates the output of the rewriter. After executing the rewritten query on the data lake, the self-checker inspects both the resulting table and the execution metadata. Based on this evidence, it decides which actions to take in the next iteration. Actions may include retrieving additional tables, searching for relevant columns, or exploring join paths between already-retrieved tables. Crucially, the self-checker can also terminate the loop once it determines that a candidate has been found that sufficiently reflects the user's intent.
- **Data Lake Actions:** A collection of specialized retrieval and transformation operations that the self-checker can invoke to support the iterative rewriting process. These actions go beyond basic table retrieval by enabling targeted searches and adaptive data transformations, for example, discovering join paths between tables, searching for specific values, or constructing cleaned versions of tables using dynamically generated UDFs. The results of these actions, such as table previews, join candidates, or normalization outputs, are stored in the scratchpad and can be reused in subsequent iterations to refine the query.
- **Scratchpad:** A shared memory that stores compact previews of tables, join relationships, CTE specifications, and UDF definitions. By preserving this context across iterations, the scratchpad

prevents redundant retrievals and ensures that the rewriter and self-checker operate on a consistent and growing body of evidence.

- **Rewrite History:** A record of all past rewrites and their execution outcomes. This history allows the system to avoid cycling between equivalent candidates and provides the rewriter with feedback on which query forms were promising or unsuccessful.

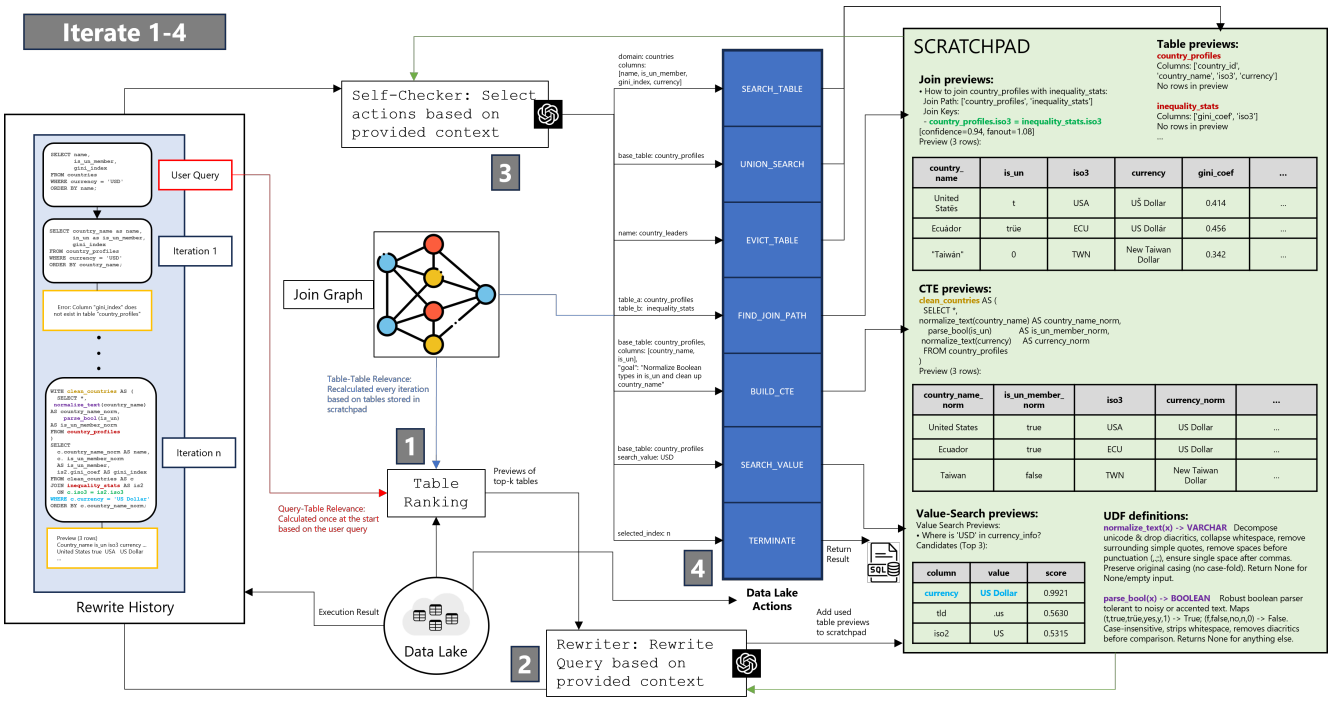

Figure 3.1: **Overview of the approach**. The visualization depicts a realistic state of how our system can look after a few iterations. It shows the different components of the approach and how they interact.

A detailed overview of this approach is shown in Figure 3.1. This visualization depicts a realistic state of how our system can look after a few iterations. Furthermore, it serves as an example to motivate the different stations of the approach. In the following sections, we describe the individual components of the system in more detail.

3.3 Data Lake Actions & Retrieval

A central element of our approach is the use of specialized data lake actions, which the self-checker can select to contribute to the iterative rewriting of the SQL query based on the outcomes of previous iterations. Each action produces structured retrieval results, such as table previews or value matches, which are stored in the scratchpad for subsequent iterations to reuse when refining the rewritten SQL query. Compared to basic table retrieval, the self-checker can use this to trigger more targeted and specialized retrievals that are directly aligned with the current information need, rather than performing broad, schema-level searches. In the following, we provide a concise overview of these actions.

- **SEARCH_TABLE(domain, columns):** This action can be used to search for specific tables similar to a provided table schema. More precisely, this table schema consists of the expected table domain and required column names. We perform a simple similarity search across the data lake to find the most promising candidates.
- **SEARCH_VALUE(base_table, search_value):** The self-checker can use this action to search for specific entries in an already retrieved table based on semantic similarity. To do so, it must provide the base table and the searched value. This retrieval mode is handy for finding the exact names of values used in filter predicates.
- **FIND_JOIN_PATH(table_a, table_b):** Assuming we lack information about foreign-key relationships, we use this action to find promising ways to join two specified tables. We do this by running a shortest-path algorithm on the Join Graph using the precomputed joinability scores. We provide more details on this in Section 5.2.
- **UNION_SEARCH(base_table):** This action is used when the self-checker identifies that the user's intent involves combining data from multiple, structurally similar tables, or if the query explicitly contains a UNION, and more compatible candidates would enrich the context. Upon invocation, the system searches for other tables in the data lake that are union-compatible with a specified base table. Compatibility is estimated using a combination of column-level semantic similarity and type matching. The top candidates are then returned along with their alignment scores and column mappings. They allow the rewriter to construct `UNION ALL` queries that merge records from different but related tables, even when schema variations or naming inconsistencies exist.
- **BUILD_CTE(base_table, columns, goal):** This action is triggered when the self-checker determines that the quality of the result is limited by noise rather than missing information, such as inconsistent date formats, diacritics, irregular boolean values, or numeric delimiters. In this case, the self-checker instructs a dedicated normalizer to build a cleaned CTE version of a given base table. To achieve this, the system dynamically generates a small set of normalization UDFs through the LLM itself rather than relying on a fixed, predefined set. This allows the UDFs to adapt to the specific noise patterns of the dataset at hand. The generated UDFs are deterministic and self-contained, handling typical cleaning operations such as Unicode decomposition, whitespace collapsing, punctuation spacing, strict boolean parsing, ISO-style date normalization, and numeric de-grouping. The resulting CTE definitions with execution previews and compact UDF metadata are stored in the scratchpad. We do not store the cleaned tables in the data lake itself to keep the query executable on the original lake. The rewriter is prompted to copy the definitions of the CTEs in the rewritten query.
- **EVICT_TABLE:** To keep the context size limited, the LLM is encouraged to flag obviously irrelevant content, which is then removed from the scratchpad. Additionally, our system also evicts content once a specific threshold is surpassed following a timestamp-based policy as described in Section 4.2.
- **OUTPUT_QUERY:** Once the self-checker identifies a rewritten query that sufficiently represents the user's intent and yields a good result, it can choose this action to terminate the entire process and return the candidate. Note that the self-checker can always choose from the whole rewrite history. For example, if it figures that the candidate produced in iteration 1 was actually a good option, it can still select it in iteration 4.

4 Iterative Query Rewriting with Self-checking

The core idea of our approach is the iterative refinement of the user query using two complementary LLMs: the rewriter and the self-checker. Together, they form a closed feedback loop that gradually transforms the user query into a form directly executable on the data lake. This section explains the details of this design, including prompt specifications and context management across iterations.

4.1 Feedback Loop

At a high level, each iteration proceeds through four phases:

1. **Proposal:** The rewriter generates a rewritten version of the query, using the current scratchpad and rewrite history as context.
2. **Execution:** Each new candidate query is executed on the data lake backend (DuckDB), yielding result previews and error traces.
3. **Evaluation:** The self-checker assesses execution results and selects data lake actions such as retrieving more relevant tables or refining join paths.
4. **Update:** The scratchpad and rewrite history are updated with new evidence, which conditions the following rewriter prompt.

This cycle repeats until the self-checker issues the `OUTPUT_QUERY` action, indicating that a sufficiently accurate and meaningful query has been obtained. Alternatively, we terminate after reaching a predefined maximum number of iterations to restrict the search space. Figure 4.1 sketches this iterative loop.

Prompting the Rewriter

The rewriter's role is generative: it produces new queries that fix execution errors or may better align with the user's intent based on the retrieved tables. To guide this process, the prompt is structured into several components:

- **Instruction:** A concise instruction framing the rewriter as a SQL rewriting assistant whose outputs must be executable and grounded in the table previews of the scratchpad. The rewritten query should fix execution errors, if present, and ensure alignment with the user's intent. We also add constraints such as "avoid repeating queries equivalent to those in history" or "rename column names after those assumed by the user".

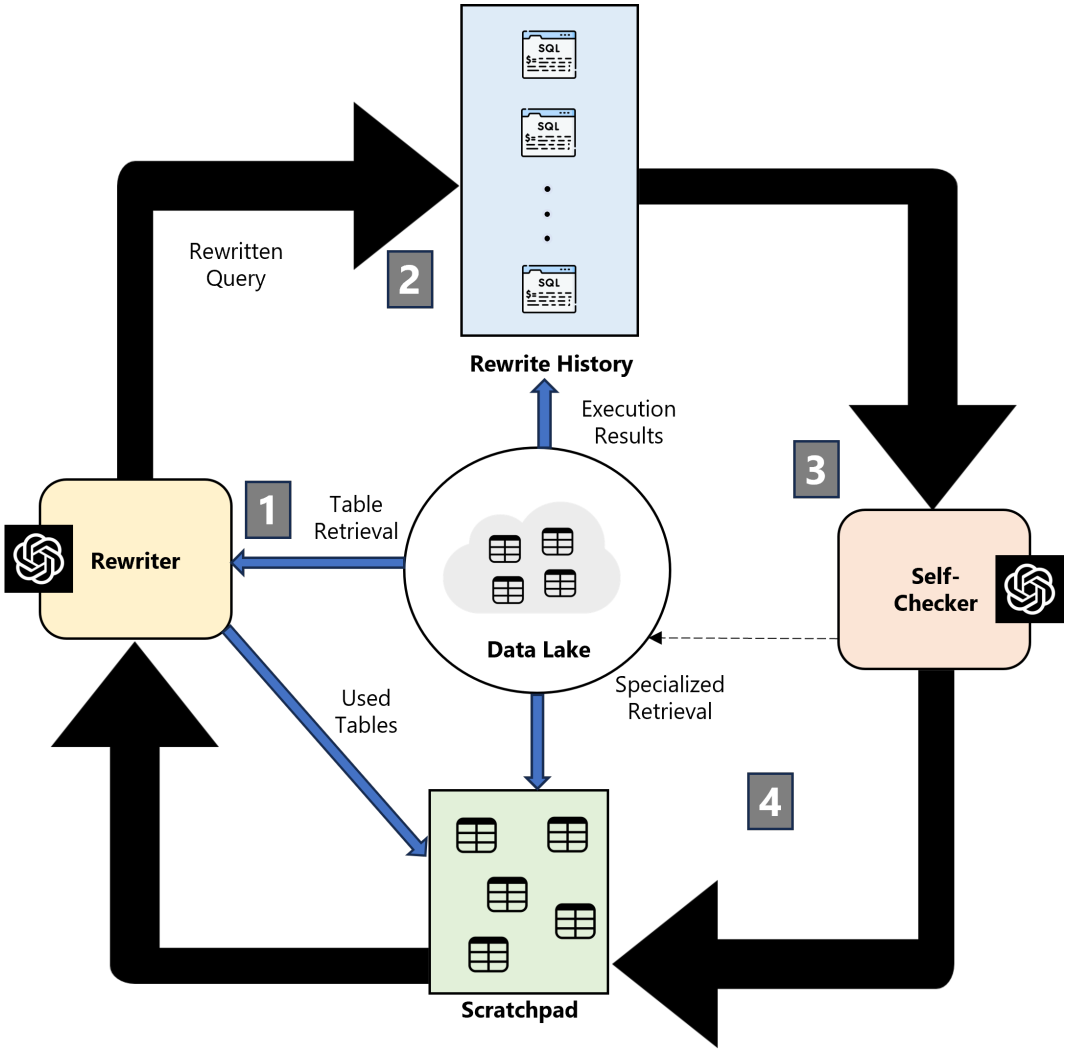

Figure 4.1: **Overview of the closed feedback loop**. The illustration shows how the rewriter and self-checker interact with the data lake, forming a closed feedback loop that iteratively refines the query.

- **Rewrite history:** Original user query and a list of previously attempted rewritten versions with their execution outcomes, including errors and table previews.
- **Scratchpad:** Compact previews of relevant tables, including example rows, summaries of discovered join paths, and CTEs with UDF definitions discovered in previous iterations.
- **Retrieved Tables:** Top-k tables retrieved in this iteration based on Query-Table Relevance and relevance to the tables stored in the scratchpad (Table-Table Relevance).

The rewriter outputs a JSON object containing the rewritten query, accompanied by a brief justification and usage metadata. The structure of a rewrite is:

```
{
  "sql": "<rewritten SQL query>",
  "reason": "<1–2 sentence rationale>",
  "used_tables": [
    {
      "table_name": "<table>",
      "columns": ["<col1>", "<col2>", ...],
      "rows": [{"<col1>": "<val1>", ...}]
    }
  ]
}
```

This prompt design enforces grounding in available evidence while still allowing the model sufficient flexibility to explore alternative query formulations. By exposing both successes and failures of prior attempts, the rewriter is effectively encouraged to perform error-driven refinement.

Prompting the Self-Checker

The self-checker prompt differs fundamentally in purpose. Rather than producing a query, it must output a structured decision about the next action. Thus, its role is discriminative instead of generative. The prompt contains:

- **Instruction:** Establishes the model as a decision-making component responsible for evaluating candidate queries and planning corrective or exploratory actions.
- **Rewrite history:** Original user query and a list of previously attempted rewritten versions with their execution outcomes, including errors and table previews. Each candidate in history is given an index to allow the LLM to reference it.
- **Scratchpad:** We also provide the scratchpad for the self-checker to improve reasoning about further actions. Note that we do not give any exploratory tables from basic table retrieval to the self-checker in comparison to the rewriter.
- **Action explanations:** Short descriptions of the actions and what parameters to provide.

The self-checker produces a structured JSON object that specifies one or more actions and their parameters, along with a compact reasoning summary justifying the decision. We also add the reasoning to the scratchpad to improve communication between agents across iterations. Note that the self-checker can return the same action multiple times with different parameters. Its structure is:

```
{
  "actions": [
    {"type": "FIND_JOIN_PATH", "table_a": "...", "table_b": "..."},
    {"type": "BUILD_CTE", "base_table": "...", "columns": [...],
     "goal": "..."},
    {"type": "SEARCH_VALUE", "search_table": "...",
     "search_value": "..."}
  ],
```

```
  "reasoning": {
    "intent_coverage": "...",
    "output_quality": "...",
    "missing_information": "...",
    "suggested_improvement": "..."
  }
}
```

The `actions` field encodes the next operation(s) the system should perform, while the `reasoning` block provides a concise justification and evaluation of the current iteration's results.

4.2 Scratchpad Management

The scratchpad is the central memory of our system to store relevant information across iterations. Instead of treating each rewriting step as an isolated event, the scratchpad, together with the rewrite history, enables a continuous refinement process in which information accumulated from earlier iterations can inform future decisions. In the following, we discuss the details of how the scratchpad is structured, how we add content to it, and how we limit its size.

Structure

The scratchpad maintains a lightweight and interpretable representation of relevant information discovered in previous iterations. It acts as a structured repository for:

- **Table previews:** Small samples of relevant tables retrieved from the data lake, allowing the rewriter to ground its understanding of schema and content. We always store all column names for each table, as it takes little space and provides a complete overview of the schema. Additionally, we offer the first rows of each table to allow insights into the actual content. The number of rows can be set as a config parameter, but usually one to three rows provide sufficient context. The tables originate either from the basic table retrieval or from the specialized data lake actions `SEARCH_TABLE` and `UNION_SEARCH`, both triggered by the self-checker. We only store tables from the basic retrieval if the rewriter uses them in a rewritten query; otherwise, they remain retrievable in later iterations but must be fetched again. However, once a table is evicted from the scratchpad, it is permanently excluded and cannot be reintroduced in subsequent iterations.
- **Join previews:** Join paths between two or more tables discovered through the `FIND_JOIN_PATH` action. For each candidate path, we store a compact preview of the resulting join and detailed path-level scores. These include metrics such as the fan-out factor, which quantifies how much a join expands the number of rows in its output relative to its inputs. Such metrics help the rewriter assess whether a join path is semantically meaningful or overly permissive, enabling more informed join decisions even in the absence of explicit foreign-key information.
- **Value Search Previews:** Results of targeted value lookups performed via the `SEARCH_VALUE` action. This action identifies where a given value (e.g., `'USD'`) or similar variants occur within the data lake by scanning column contents of specified tables. For each lookup, we record the top 3 matching occurrences based on embedding similarity, including the table and column names and example rows

containing the match. These previews help the rewriter resolve ambiguous filter predicates or align inconsistent value representations across tables.

- **UDF metadata:** Overview of LLM-generated UDFs that handle data cleaning or parsing tasks. More precisely, we store the UDF's signature, its return type, and a short description explaining its purpose.
- **CTE previews:** List of CTEs to clean noisy tables using the LLM-defined UDFs. Moreover, we provide previews of the tables created by executing the CTEs. We do not add these tables to the data lake itself to keep the query executable on the original lake. The rewriter is prompted to define the CTEs in the rewritten query explicitly.

An example state of the scratchpad is visualized in Figure 3.1.

Eviction policy.

To make our approach scalable, it is essential to limit the size of the scratchpad. Otherwise, the context provided to the LLM would grow too large, potentially hurting the result and increasing costs. We provide two methods to keep the scratchpad size tractable:

- **Time-based Eviction:** Once the size of a scratchpad section exceeds a threshold, we evict content according to a timestamp-based policy. When we add to the scratchpad, each entry gets a timestamp set to the time of insertion. We want to encourage exploration by evicting the oldest entries once the context grows too large. Note that evicted information may still appear in the rewrite history. With this in mind, our policy aims to remove entries that were either already used and displayed in the rewrite history or turned out to be less valuable. The maximum number of entries per section can be set as a config parameter.
- **EVICT_TABLE action:** To support the eviction process, we also provide the self-checker with the option to flag scratchpad entries as irrelevant. We encourage the LLM to choose this action only in clear cases, as evicting the wrong table too early can significantly affect the outcome of our approach.

Once a table is evicted, it is excluded from the basic table retrieval to avoid repeating actions. However, it can still be accessed via specialized data lake actions, as evicted tables may remain relevant, for instance, when they serve as intermediate nodes in the join path between two other tables.

5 Finding Relevant Tables

A central component of our architecture is the table retrieval process, which identifies the most relevant tables from the data lake in each iteration to ground the query rewriting process. This section presents our basic retrieval mechanism, executed once per iteration, which ranks tables based on *Query–Table Relevance* and *Table–Table Relevance*. In addition, we describe the specialized *data lake actions* that allow the self-checker to refine retrievals to meet more specific information needs, such as searching for missing attributes, resolving filter values, or discovering joinable tables. Together, these retrieval strategies form the foundation for iteratively aligning the user's intended query with the data lake's actual structure and content.

5.1 Basic Table Retrieval

The table retrieval module retrieves relevant tables from the data lake in each iteration. Unlike the specialized data lake actions selected by the self-checker, this retrieval is performed automatically in each iteration. Based on a ranking explained in the following sections, we provide the top-k retrieved tables to the rewriter. They can then be used to map the user query, which is based on the user's assumed schema, to the actual data lake schema. With this in mind, the role of table retrieval in the approach is exploratory, providing new evidence in each iteration. In the following, we discuss the actual metrics used to rank table relevance.

Challenges

When rewriting user queries, we face three recurring challenges:

1. **Wrong column/table names, but correct data split.** The user query may reference non-existent names (e.g., `author_id` instead of `writer_id`), while still assuming the correct division of data across tables. Here, the retrieval module must identify the correct schema elements through similarity of names and semantics.
2. **Wrong idea of data partitioning.** In other cases, the query assumes that all information is contained in a single table, when in reality it is distributed across multiple tables (or vice versa). Specialized data lake actions must then guide the rewriting process towards the correct decomposition or merging of sources.
3. **Join discovery without explicit keys.** Since our data lake lacks foreign key annotations, we cannot rely on explicit constraints. Instead, we use a precomputed *Join Graph*, which estimates the likelihood that two tables can be joined based on schema similarity and content statistics.

The first type of mismatch is largely handled by our Query-Table Relevance-based retrieval since pure similarity is sufficient to discover the relevant tables. On the other hand, we try to handle the third type with Table-Table Relevance-based retrieval and dedicated `FIND_JOIN_PATH` actions. Lastly, the second source of errors must be handled by the LLM agents. The rewriter must detect the correct data partitions using the provided context, whereas the self-checker must identify the missing parts of the data to search for.

5.1.1 Query–Table Relevance.

In the first iteration, our system computes a Query-Table Relevance score between the user query and every table in the data lake. To do so, we first create concise string representations of both the data lake tables and the tables mentioned in the query. Each table mentioned in the query is represented as a concise text snippet formed by concatenating its assumed table name with the list of referenced column names. Moreover, we emphasize the column tokens by repeating them multiple times. On the other hand, for regular tables, we concatenate table and column names to illustrate the schema. Furthermore, we normalize all table and column names by converting them to lowercase, decomposing underscores and camel-case into natural-language tokens, removing punctuation and file extensions, collapsing multiple spaces, and applying Unicode normalization to strip diacritics. This ensures that variations such as `Gini_Index`, `giniIndex`, and `Gíni Index` are all mapped to the unified form `gini index`, allowing consistent embeddings across heterogeneous naming conventions. We compute one embedding per table mentioned in the query, and for every table in the data lake. Figure 5.1 provides an example of this approach. Both representations are embedded using a shared embedding model, and their cosine similarity is computed to estimate semantic alignment. We use the all-MiniLM-L6-v2 model from SentenceTransformers (Reimers and Gurevych 2019), which produces 384-dimensional embeddings trained on large-scale semantic similarity datasets, offering a compact yet robust representation well suited for table- and query-level retrieval. The Query-Table Relevance of a candidate table t_i is calculated as:

$$R_Q(t_i) = \max_{q_j \in \mathcal{Q}} \text{sim}\big(E(t_i), E(q_j)\big),$$

where $E(\cdot)$ denotes the embedding function based on the all-MiniLM-L6-v2 model, $\text{sim}(\cdot,\cdot)$ represents cosine similarity, and $\mathcal{Q}$ is the set of query-derived table snippets. The resulting value $R_Q(t_i)$ measures how closely the schema and semantics of the table t_i align with the user's intended query structure.

The resulting similarities form the initial ranking list. This idea follows the concept of Query-Table Relevance described by Chen et al. (2025). Note that we do not need to recompute these scores in later iterations, since both the user query and the tables in the lake are static.

5.1.2 Table–Table Relevance.

In every iteration after the first, we rerank the tables based on the previously retrieved tables currently in the scratchpad. We do so by considering Table-Table Relevance. In our use case, this metric measures the compatibility of tables with the tables already stored in the scratchpad. More precisely, this compatibility

Figure 5.1: **Overview of Query-Table Relevance calculation**. The visualization shows how the system embeds the tables mentioned in the user query and the tables in the data lake to compute their semantic similarity. This enables identifying the relevant tables for rewriting the query.

is measured by using the precomputed joinability scores stored in the Join Graph. Formally, for each candidate table t_i, we define its score in iteration k as:

$$S_k(t_i) = R_Q(t_i) + \max_{t_j \in \mathcal{C}_k} \Omega(t_i, t_j),$$

where $R_Q(t_i)$ is the static Query-Table relevance from the first iteration, $\Omega(t_i, t_j)$ is the joinability score between tables t_i and t_j, and $\mathcal{C}_k$ is the set of context tables stored in the scratchpad at iteration k.

This hybrid ranking strategy allows the retrieval process to become increasingly join-aware over time, following a similar intuition as Chen et al. (2025)'s join-aware multi-table retriever.

We exclude tables from the ranking once they are added to the scratchpad. Moreover, we do not reconsider them once they are evicted from it. This mechanism resembles the removal step in the MURRE framework by Zhang et al. (2024), which discards already used tables from the candidate pool to promote diversity and prevent overspecialization.

5.2 Estimating Joinability with Join Graphs

One of the core constraints in our setup is the lack of foreign-key information. To deal with this, we use a Join Graph to estimate the Joinability between tables in the data lake. In the following sections, we will explain the construction of the graph, how we find join paths between two given tables, and how we identify candidate join keys.

5.2.1 Building the Join Graph

Each node in the Join Graph represents a table in the data lake. An undirected edge is added between two tables if they share at least one column pair whose values overlap and appear semantically related. To quantify this relationship, we compute the following scores between every table pair:

1. **Column-level similarity:** For every column pair (c_a, c_b) between two tables t_i and t_j, we compute a combined schema- and instance-based similarity

$$S(c_a, c_b) = \big(S_{\text{schema}}(c_a, c_b) + S_{\text{inst}}(c_a, c_b)\big) \cdot K(c_a, c_b),$$

   where $S_{\text{schema}}(c_a, c_b)$ is the cosine similarity between their column embeddings, $S_{\text{inst}}(c_a, c_b)$ is the Jaccard similarity between their value sets, and $K(c_a, c_b)$ is the maximum uniqueness of the two columns. The uniqueness factor is used to encourage joins between two columns, where at least one of them is a primary key, to improve the quality of the join.

2. **Table-level joinability:** The joinability score between two tables t_i and t_j is then defined as

$$\Omega(t_i, t_j) = \max_{c_a \in t_i,\, c_b \in t_j} S(c_a, c_b),$$

   which represents the strongest evidence of a potential join between the two tables, based solely on their schema- and instance-level similarity.

This approach follows the Table–Table Relevance formulation proposed by Chen et al. (2025). An undirected edge between t_i and t_j is added if at least one column pair yields a positive instance similarity, indicating potential value overlap. Each edge stores the overall joinability score $\Omega(t_i, t_j)$ together with the top-k column pairs and their individual similarity scores. The graph is then saved in a compact JSON representation and can be accessed later in the approach. Note that the graph is not dependent on the actual user query and can therefore be used independently.

5.2.2 Querying the Join Graph

In our approach, we utilize the Join Graph in two different scenarios: upon invocation of `FIND_JOIN_PATH` and in the Table-Table Relevance-based table reranking in every iteration.

Using the Join Graph to find Joinable Tables In each iteration, we use the Join Graph to identify new tables that are likely joinable with the current context stored in the scratchpad. Each node in the graph represents a table, and each edge encodes the precomputed joinability score $\Omega(t_i, t_j)$ between two tables. For every candidate table t_i that is not yet part of the scratchpad, we compute a hybrid score combining its semantic relevance to the query with its structural compatibility to the already selected context tables:

$$S_k(t_i) = R_Q(t_i) + \max_{t_j \in \mathcal{C}_k} \Omega(t_i, t_j),$$

where $R_Q(t_i)$ is the static Query–Table relevance computed in the first iteration, $\Omega(t_i, t_j)$ is the joinability score between t_i and t_j, and $\mathcal{C}_k$ is the set of context tables currently in the scratchpad. This direct joinability ranking allows the retrieval to become increasingly join-aware in later iterations, preferring tables that are

not only semantically relevant to the query but also strongly connected to the existing context through high joinability edges in the Join Graph.

Using the Join Graph to predict Join Paths Whenever the required data is split across multiple tables, we have to find a way to connect them without explicit foreign-key information. We model this problem as a shortest-path problem on the Join Graph. The goal is to find the shortest path between the two nodes that represent the tables we want to join. First, we need to define the cost function we want to maximize. A naive approach is to simply use the joinability score between two tables as the cost of traversing between them. However, this approach has the drawback that we may favor paths that contain some very strong joins, even if they also include relatively weak ones. Therefore, we have to punish the choice of weak joins more aggressively. One option is to make the path-cost calculation multiplicative rather than additive. The idea is to maximize

$$\prod_{k=1}^{L} \Omega(t_{k-1}, t_k)$$

where L is the number of edges along the path.

The problem is that efficient shortest-path algorithms such as Dijkstra's or A* assume additive path costs and therefore cannot directly handle multiplicative objectives. To deal with this, we apply a logarithmic transformation that converts the product into a sum. Furthermore, we negate the logarithmic term to transform the maximization objective into a minimization problem, allowing standard shortest-path algorithms to be applied directly:

$$C(P) = -\log\left(\prod_{k=1}^{L} \max(\Omega(t_{k-1}, t_k), \varepsilon)\right) + \lambda L = -\sum_{k=1}^{L} \log\big(\max(\Omega(t_{k-1}, t_k), \varepsilon)\big) + \lambda L,$$

where $\lambda \geq 0$ is a fixed hop penalty that discourages unnecessarily long paths and ε is a small constant to avoid taking the log of 0. This transformation makes the total path cost additive, allowing us to apply standard shortest-path algorithms while still reflecting the multiplicative nature of join reliability so that a single weak join has a strong negative effect on the overall path score. For our use case, it is convenient to find the top-k shortest paths rather than just the shortest one. We use Yen's algorithm on the Join Graph with the proposed cost function to achieve this (Yen and YENt 2007). Next, we have to predict the join keys for a given path. To do so, we use the top-k join key candidates based on the column-level scores between each table of the path, stored in the Join Graph. Furthermore, we apply an additional selection based on actually executing the join on the data. More precisely, for each join key, we calculate its fan-out factor $f(c_a, c_b)$, defined as the average number of matching rows in the right table per distinct key value in the left table:

$$f(c_a, c_b) = \frac{\text{rows}_{\text{joined}}}{\text{distinct}_{\text{keys}}},$$

where $\text{rows}_{\text{joined}}$ is the number of resulting rows after joining on the key pair (c_a, c_b), and $\text{distinct}_{\text{keys}}$ is the number of unique key values in the left table. A high fan-out indicates a non-selective join, which we want to penalize.

We then compute a final join quality score that balances schema-level similarity and join selectivity:

$$S_{\text{join}}(c_a, c_b) = \frac{S(c_a, c_b)}{(1 + f(c_a, c_b))^{\alpha}}.$$

where $S(c_a, c_b)$ is the column-level similarity between c_a and c_b. The parameter α controls how strongly high fan-out joins are down-weighted. The resulting score S_{join} serves as the final ranking criterion for selecting the best join key candidates along each path. Subsequently, we execute the proposed join paths and provide execution previews and scores to the scratchpad. Figure 5.2 illustrates this process.

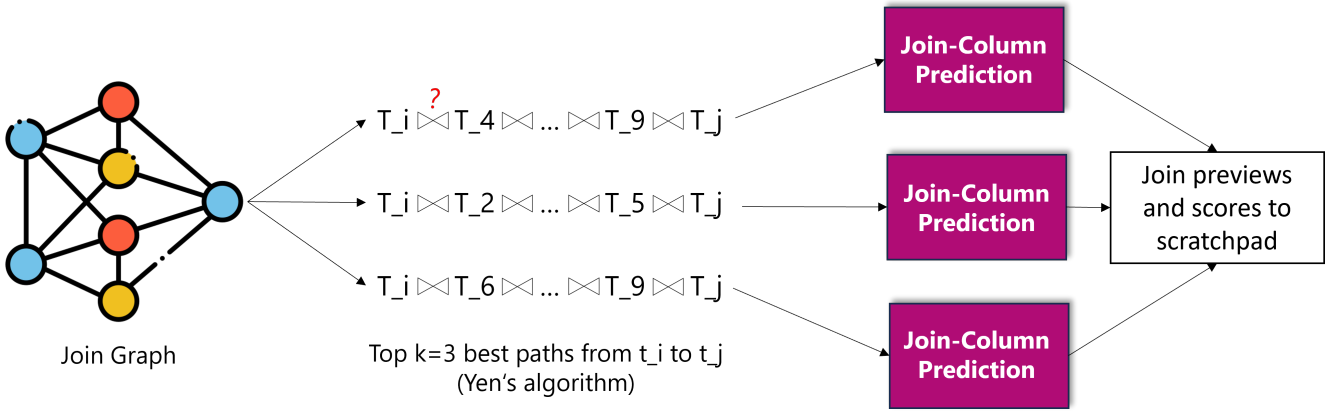

Figure 5.2: **Overview of the process to find join paths**. The illustration shows how the system uses the Join Graph to identify the top-k most reliable join paths between two tables using Yen's algorithm and join-column prediction.

5.3 Specialized Similarity Retrieval

The system offers three specialized data lake actions that allow more tailored retrieval based on semantic similarity: `SEARCH_TABLE`, `SEARCH_VALUE`, and `UNION_SEARCH`. In contrast to basic table retrieval, which ranks candidates solely by overall schema similarity, these specialized actions enable more fine-grained and context-aware semantic searches that focus on specific missing aspects identified by the self-checker.

Search_Table. The `SEARCH_TABLE` action is triggered when the self-checker detects that the current rewritten query references columns that do not exist in the working set. For example, this can be used to recover from an execution error, such as the one shown in Section 3.1, where the query failed because the column `gini_index` was missing. In such a case, the self-checker formulates a targeted retrieval request specifying the missing column name and the general domain of the query (e.g., `countries`). Given these hints, the system constructs an embedding-friendly snippet that combines both the domain and the required column names:

$$\text{snippet}(T, C) = \text{“col}_1\ \text{col}_1\ \text{col}_2\ \text{col}_2\ \ldots\ \text{table”}.$$

This representation emphasizes column semantics while retaining the table context. The snippet is embedded and compared via cosine similarity to cached table embeddings built from schema snippets of the form

$$\text{schema}(t) = \text{“table column}_1\ \text{column}_2\ \ldots\text{”}.$$

Tables with the highest embedding similarity are returned as candidates, enabling the system to automatically discover semantically relevant sources that are likely to contain the missing attributes.

Search_Value. The `SEARCH_VALUE` action focuses on cell-level similarity within a known table. It is typically used when the self-checker suspects that a `WHERE` or `JOIN` condition relies on an incorrect or mismatched value. For instance, if the query filters for 'USD' but the relevant column contains a slightly different version, such as 'US Dollar', the self-checker can perform a value search to identify the closest matching entries. In this step, the retriever embeds the target phrase and compares it to embeddings of distinct cell values across all columns in the given table. For efficiency, columns are limited to a fixed number of distinct values (typically 2 000), and results are ranked by a weighted score combining:

- semantic similarity between the value and each cell, and
- a substring or exact-match bonus for literal overlap.

The results inform subsequent rewriting iterations by revealing which columns most likely contain the intended entity or category, enabling the system to refine filter predicates or join keys based on data content rather than column names alone.

Union_Search. The `UNION_SEARCH` action is triggered when the self-checker detects missing records or an explicit `UNION` request. The system then locates semantically compatible tables that can be vertically combined to extend the current result. To assess compatibility, the system infers coarse type families for each column (numeric, boolean, datetime, or string) directly from sampled cell values using lightweight heuristics based on value patterns and parsing success rates. Column alignment between the base and candidate tables is determined through greedy bipartite matching over column-name embeddings, prioritizing high semantic similarity and consistent inferred types. Each candidate receives a unionability score that integrates both semantic and structural evidence:

$$\text{score} = 0.6 \cdot \text{mean\_sim} + 0.4 \cdot \text{coverage} + \text{type bonus},$$

where mean_sim captures the average semantic similarity between aligned columns, coverage reflects the proportion of columns that could be matched between the two tables, and the type-consistency bonus rewards candidates with a high fraction of type-compatible matches. The top-ranked tables are stored in the scratchpad, along with an annotation indicating they are potential union partners for the corresponding base table.

6 Noise Handling with CTEs and UDFs

One of the core challenges when working with data lakes is the presence of noisy or inconsistent data. Such noise can stem from formatting errors, inconsistent categorical values, or variations in text and encoding. When querying structured data with SQL, it is crucial to ensure that the results are high-quality and faithfully reflect the query's intent. In the following section, we describe how our system uses the self-checker mechanism to automatically detect data noise and employs LLM-defined UDFs to clean the affected data. A key strength of our approach lies in its flexibility: rather than relying on a fixed set of predefined cleaning functions, it dynamically generates tailored UDFs that adapt to the specific type of noise in the data.

6.1 Detecting Noisy Data

Whenever the self-checker notices noisy data in the provided table previews of the scratchpad, it can use the `BUILD_CTE` action to start the cleaning process. Distinguishing regular and noisy data is a core challenge in this process. For our purpose, we specifically prompt the self-checker to return this action only if the data is clearly noisy. More subtle inconsistencies are ignored to prevent false positives. To support this detection, we provide the self-checker with typical examples of noisy data, including encoding inconsistencies, malformed Unicode characters, irregular whitespace, and mixed text casing. These examples help recognize common error patterns. The self-checker must then provide the relevant tables and columns, together with a short description of the detected noise, to use `BUILD_CTE`.

6.2 Using LLM-defined UDFs to Clean Data

Upon invocation of `BUILD_CTE`, a separate LLM is called to clean the specified data. We decided to outsource this task to provide additional context. More precisely, instead of using the small scratchpad table previews as context, we create larger previews with up to 50 rows to enhance data coverage. The LLM then proposes a small set of Python UDFs together with a single CTE that applies them. Conceptually, the model is asked to identify concrete noise patterns observable in the preview and design narrowly scoped, data-specific UDFs to fix them. Typical outcomes include aligned date formats, canonicalized booleans, trimmed and de-duplicated whitespace, removal of markup/control characters, and harmonized categorical spellings. We express the cleaned data as a CTE layered on top of the original table, so the raw data remains intact and the transformation is auditable.

6.3 Storing Cleaned Data

The resulting UDFs are compiled and registered in the data lake backend, allowing the rewriter to apply them in subsequent query rewrites without additional model calls. Moreover, the CTEs, along with small previews of their execution results, are stored in the scratchpad. That way, the rewriter has an overview of the cleaned data versions without having to add the updated tables to the actual data lake. Note that the rewriter is prompted to define the CTEs in the returned query explicitly. Additionally, the self-checker can use the CTE previews to refine the cleaning process with other tailored `BUILD_CTE` calls in further iterations. An overview of the entire data cleaning pipeline is shown in Figure 6.1.

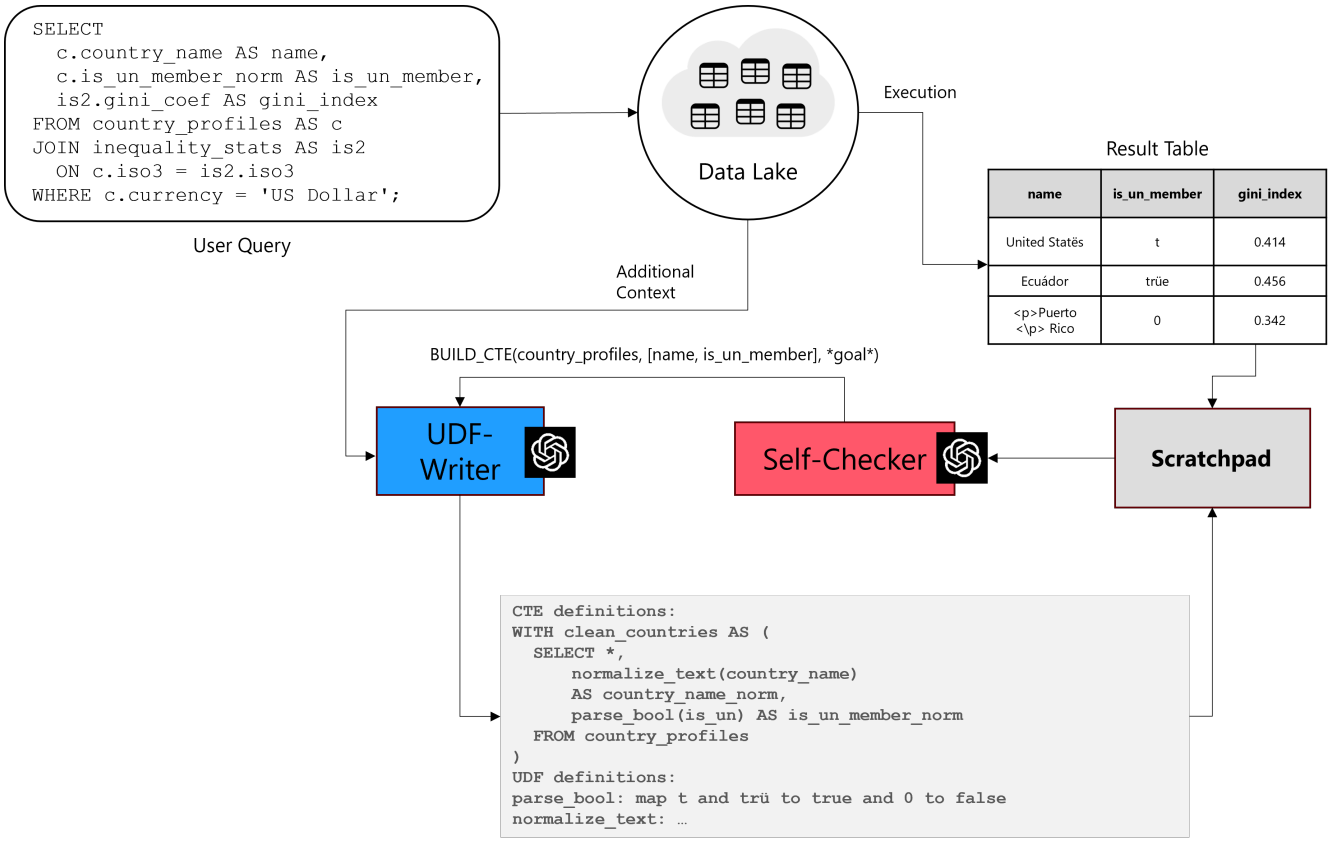

Figure 6.1: **Overview of the data cleaning pipeline**. The illustration shows how the self-checker detects noisy values and triggers the `BUILD_CTE` action to clean the data using LLM-defined UDFs.

7 Evaluation

To evaluate our approach, we execute a set of hand-written queries on a data lake derived from WikiDBs. Each query contains some type of schema mismatch, like different naming or missing tables, to represent incorrect assumptions about the schema made by the user. Additionally, we provide gold versions of the queries to compare the results produced by our approach. In the following section, we offer a detailed overview of the evaluation dataset and explain the baselines we use to show the advancements of our approach. Afterwards, we describe the experimental setup and analyze our results.

7.1 Dataset

The data lake we use to evaluate our approach stems from WikiDBs. WikiDBs is a large-scale corpus containing 100,000 relational databases from various domains. It is based on data from Wikidata. Each database consists of multiple tables that are connected by foreign keys (Vogel et al. 2024). Our approach does not have access to the provided foreign-key information to create a realistic data lake environment. The database we chose to build the data lake consists of 83 tables, each with up to 366 columns and 474 rows. Moreover, the database includes records of countries, people, companies, and universities with some overlap across tables. The tables we find in data lakes are usually created in isolation, hence this overlap depicts a realistic characteristic. For the evaluation, we create pairs of user queries and gold queries that show how our system should rewrite the query to work on the data lake and properly reflect the user's intent. Initially, we handcrafted a set of 30 queries that cover the different difficulties of our problem. These difficulties include incorrect naming of tables, columns, or entries, and wrong decompositions of data across tables. We assume that the user does not make syntactic errors. Subsequently, we use GPT-5 mini to automatically generate similar pairs based on the provided examples. To make sure they have sufficient quality, we also manually test these generated queries. In addition to further hand-crafted queries, our dataset contains 103 pairs. Additionally, we categorize the query pairs by difficulty (easy, medium, hard) and annotate both the types of incorrect assumptions made by the user and the presence of noisy data. We distinguish between seven difficulty types that capture specific sources of error in user queries:

- **Name Mismatch:** The user refers to tables or columns using names that differ from those in the data lake schema (e.g., `country` vs. `nation`).
- **Missing Table:** The query omits one or more tables required to answer the question correctly.
- **Obsolete Table:** The user references an irrelevant table that can be omitted.
- **Key Mismatch:** A join is performed on incorrect key columns.
- **Filter Mismatch:** The query applies incorrect filter conditions (e.g., wrong value range or predicate).

- **Union:** The query needs to combine semantically related tables through a union.
- **Noise:** The data itself contains inconsistencies such as irregular encodings, spelling variations, or mismatched categorical values. We manually injected such noise into a subset of tables, and only queries tagged with this difficulty are executed on the noisy version of the data lake.

Note that one query can contain multiple of these difficulty types. Figure 7.1 shows an example query pair in that format.

```
QueryResultPair(
    user_query="""
    SELECT DISTINCT
        name,
        life_summary,
        place_of_birth,
        place_of_death,
        graveyard,
        burial_country,
        grave_type
    FROM
        compositions
    ORDER BY
        name
    LIMIT 1;
    """,

    golden_query="""
    SELECT DISTINCT
        full_name AS name,
        biography AS life_summary,
        birth_location AS place_of_birth,
        death_location AS place_of_death,
        burial_site AS graveyard,
        country AS burial_country,
        burial_category AS grave_type
    FROM
        composer_details
    LEFT JOIN
        notable_musical_works
        ON composer_details.full_name = notable_musical_works.composer
    LEFT JOIN
        place_of_burial
        ON composer_details.burial_site = place_of_burial.cemetery_name
    ORDER BY
        full_name
    LIMIT 1;
    """,

    result_table=pd.DataFrame([
        {
            "name": "Alexander Vasilyevich Alexandrov",
            "life_summary": "Russian Soviet composer",
            "place_of_birth": "Plakhino",
            "place_of_death": "Berlin",
            "graveyard": "Novodevichy Cemetery",
            "burial_country": "Russia",
            "grave_type": "Category:Burials at Novodevichy Cemetery"
        }
    ]),

    difficulty_tags=["Missing Table", "Name Mismatch"],
    difficulty="medium",
    required_tables=[
        "composer_details",
        "notable_musical_works",
        "place_of_burial"
    ]
)
```

Figure 7.1: **Example of a `QueryResultPair` used in our evaluation dataset**. It contains the user query, the correct rewritten (gold) query, and the resulting table. Difficulty tags describe the kinds of incorrect user assumptions (e.g., missing table, name mismatch).

7.2 Baselines

To put our work into perspective, we created two baseline approaches to our problem. First, we deploy a Retriever-Reader baseline that retrieves relevant tables for the user query and is asked to directly return the table that answers the query, without any query rewriting or execution. Secondly, we design a basic Retriever–Rewriter Baseline that uses the relevant tables to rewrite the query and returns the execution results directly without further iterations. Figure 7.2 and Figure 7.3 give an overview of both approaches. For both baselines, we use a ChromaDB-based retriever that embeds each table and performs semantic similarity search to identify the most relevant tables for a given user query. Compared to our main approach, we cannot consider any Table-Table Relevance, since both baselines work in a single iteration. The prompts for the baseline approaches include only a short description of the task, the user query, and previews of the top-k semantically most relevant tables. Note that the Retriever-Reader baseline requires substantially larger table previews to produce meaningful results, as the LLM must generate the entire output table itself and therefore needs all relevant rows available in its context. Compared to our main approach, the Retriever–Rewriter baseline also depends on more extensive table previews when correcting incorrect filter predicates, as it needs access to specific table entries rather than just the schema. While our main approach can explicitly look up such values using the `SEARCH_VALUE` action, the Retriever–Rewriter baseline must rely solely on the information contained within its previews. For our evaluation, we provide the Retriever-Reader with the first 20 rows from each table and the Retriever–Rewriter with the first 10. Both the Retriever-Reader and Retriever–Rewriter baselines use the same underlying model, GPT-5 mini, as employed in our main approach.

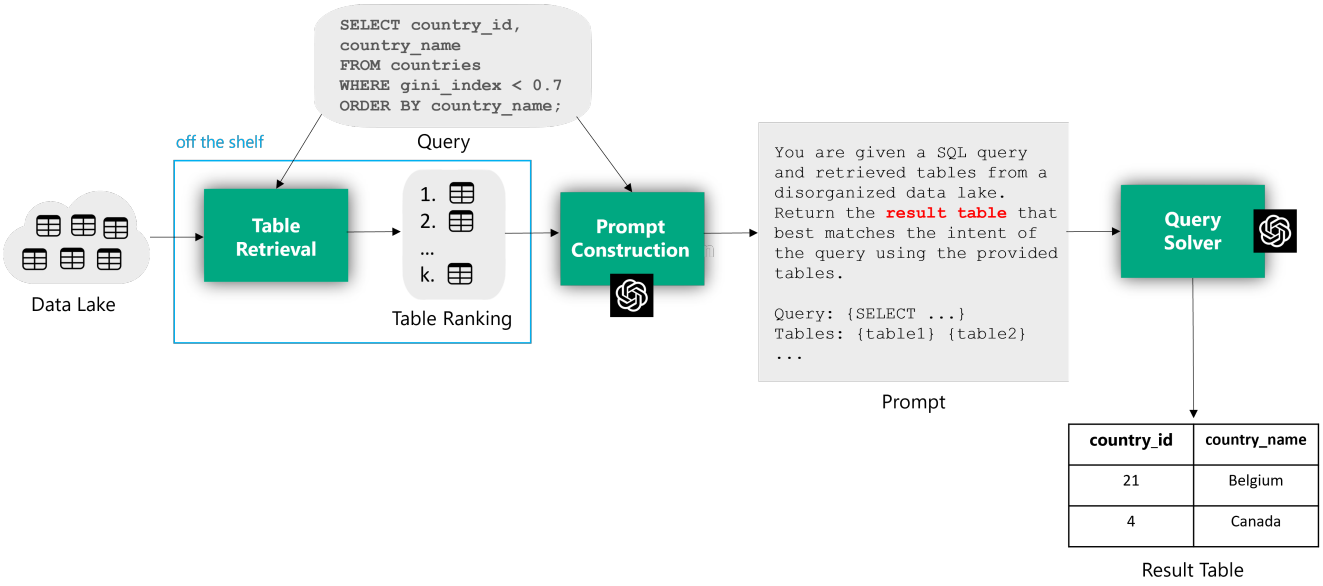

Figure 7.2: **Overview of the Retriever-Reader Baseline**. It retrieves relevant tables for the user query and is asked to directly return the table that answers the query, without any query rewriting or execution.

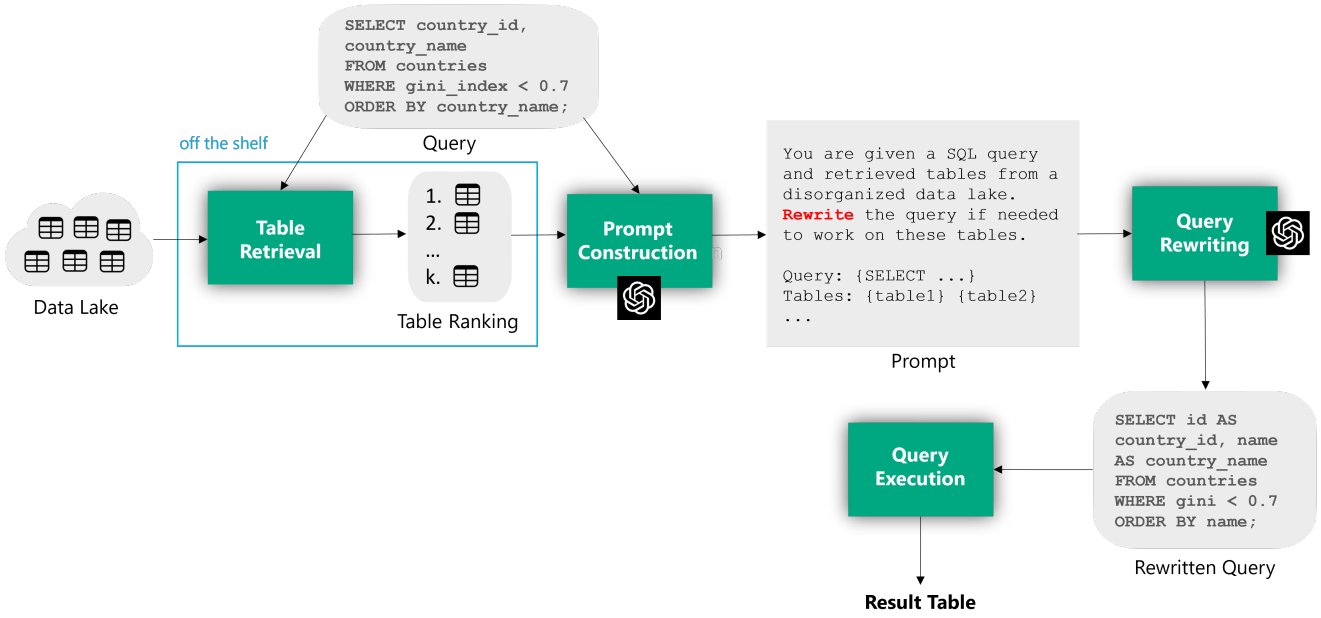

Figure 7.3: **Overview of the Retriever-Rewriter Baseline**. It retrieves relevant tables to rewrite the query and returns the execution results directly without further iterations.

7.3 Experimental Setup

All experiments are conducted on the data lake introduced in Section 7.1, covering multiple domains, including countries, people, companies, and universities. For all methods, we employ the same underlying embedding model, all-MiniLM-L6-v2, to embed tables into a ChromaDB index for semantic retrieval. The default language model used throughout all experiments is GPT-5 mini.

Retrieval differences. While all systems rely on the same ChromaDB index for table embeddings, they differ substantially in how they perform retrieval. The *Retriever-Reader* and *Retriever-Rewriter* baselines use a single-shot semantic retrieval of the top-k tables based purely on embedding similarity between the user query and each table. The retrieved tables are included in the LLM prompt as fixed previews (Retriever-Reader: first 20 rows; Retriever–Rewriter: first 10 rows), and no further retrieval or refinement occurs. In contrast, our main iterative approach employs a hybrid retrieval strategy that combines semantic similarity with structural cues, such as column-name overlap, joinability scores from a precomputed Join Graph, and value-based lookups. This hybrid retriever operates in multiple iterations and can dynamically invoke specialized search actions such as `SEARCH_TABLE`, `FIND_JOIN_PATH`, `UNION_SEARCH`, and `SEARCH_VALUE` based on the self-checker's reasoning.

Evaluation procedure. We evaluate all methods on the 103 `QueryResultPair` examples from our dataset, each containing a user query, its correctly rewritten (gold) version, and the corresponding result table. For each prediction, we first measure the *execution accuracy*, which is considered correct only if the

predicted table exactly matches the expected (gold) table in both content and, when an `ORDER BY` clause is present, in row order. However, since this criterion is strict and does not reward partially correct outputs, we additionally employ more fine-grained metrics that capture structural and content-level similarity between predicted and gold tables. The following metrics are used for this purpose:

1. **Column-level F1-score:** This metric measures how well the predicted table T_p reproduces the correct column contents of the gold table T_g. Let C_g denote the set of column names in T_g. For each column $c \in C_g$, we extract its cell values as a list of strings from both tables, denoted by $V_p(c)$ for T_p and $V_g(c)$ for T_g. If column c is missing in T_p, its precision and recall are set to zero. Otherwise, precision and recall for that column are defined as

$$P_c = \frac{|\{v \in V_p(c) \mid v \in V_g(c)\}|}{|V_p(c)|}, \qquad R_c = \frac{|\{v \in V_g(c) \mid v \in V_p(c)\}|}{|V_g(c)|}.$$

   The overall column-level precision, recall, and F1-score are the averages across all gold columns:

$$P_{\text{col}} = \frac{1}{|C_g|} \sum_{c \in C_g} P_c, \quad R_{\text{col}} = \frac{1}{|C_g|} \sum_{c \in C_g} R_c, \quad F1_{\text{col}} = \frac{2 \cdot P_{\text{col}} \cdot R_{\text{col}}}{P_{\text{col}} + R_{\text{col}}}.$$

   Column-level precision therefore captures how many of the predicted cell values appear in the gold column, while recall measures how many of the gold cell values are present in the prediction.

2. **Row-level F1-score:** To assess the correctness of the returned tuples, we compare the predicted rows and gold rows on their overlapping columns. Let R_p and R_g be the sets of rows (tuples) from T_p and T_g, respectively, and let $\text{match}(r, R)$ return the maximum fraction of columns in which row r exactly matches any row in R. Row-level precision, recall, and F1-score are defined as

$$P_{\text{row}} = \frac{1}{|R_p|} \sum_{r_p \in R_p} \text{match}(r_p, R_g), \quad R_{\text{row}} = \frac{1}{|R_g|} \sum_{r_g \in R_g} \text{match}(r_g, R_p), \quad F1_{\text{row}} = \frac{2 \cdot P_{\text{row}} \cdot R_{\text{row}}}{P_{\text{row}} + R_{\text{row}}}.$$

   This formulation allows partial credit for rows that match only some columns, capturing partially correct outputs rather than requiring binary matches.

3. **Final Score:** The final evaluation score is computed as the mean of the column- and row-level F1-scores:

$$F1_{\text{final}} = \frac{1}{2}\big(F1_{\text{col}} + F1_{\text{row}}\big).$$

   This metric jointly captures both structural correctness and content correctness. For simplicity, we refer to this metric as F1 in the following. If the query includes an `ORDER BY` clause, F1-scores are still computed independently of ordering to fairly reflect partial correctness.

In addition, we record the total LLM cost in USD and log all intermediate query traces, rewritten SQL candidates, and result tables to enable detailed inspection and reproducibility. The maximum number of iterations for the Iterative Rewriter is set to five.

7.4 Results

This section presents our results and highlights our main findings. Following the evaluation dimensions introduced in Section 7.3, the first subsection compares the performance of our approach against the baseline models, while the second subsection investigates the impact of individual system components through ablation studies. Afterwards, we conduct a detailed system behavior analysis to understand how the individual components interact during iterative rewriting. This analysis examines how the model discovers required tables over multiple iterations, how frequently specific actions are invoked, and how effectively the self-checker selects the best candidate queries.

7.4.1 Comparison to Baselines

To evaluate the effectiveness of our iterative SQL rewriting framework, we compare it against two baseline systems: (1) a Retriever-Reader model that directly predicts the result table from retrieved table previews, and (2) a Retriever-Rewriter baseline that generates a single-step executable SQL query without further refinement. All methods are evaluated on the same 103 query–result pairs using the WikiDBs-based data lake described in Section 7.1, ensuring consistent conditions across approaches.

Overall Performance. Figure 7.4 summarizes the overall performance of all three systems. Our proposed Iterative Rewriter achieves an F1 of 0.65, clearly outperforming both the single-step Retriever–Rewriter baseline (0.41) and the Retriever-Reader baseline (0.28). This represents an absolute improvement of +0.24 F1 over the Retriever–Rewriter and 0.37 F1 over the Retriever-Reader.

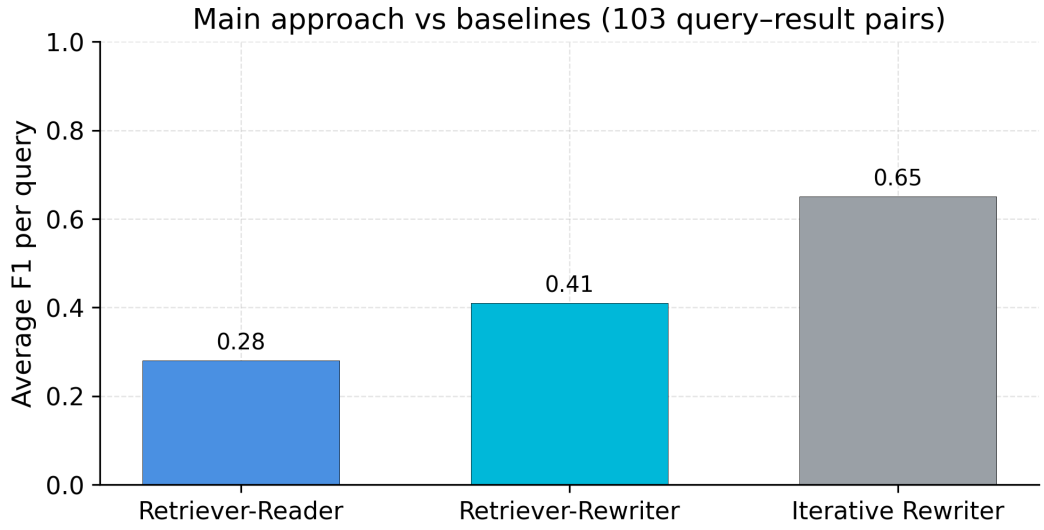

Figure 7.4: **Overall comparison of F1 scores**. The illustration shows the average F1 scores per query of the baseline approaches (Retriever–Reader and Retriever-Rewriter) compared to the proposed Iterative Rewriter approach across all 103 query–result pairs.

Execution Accuracy. In addition to the F1 metric, we also measure strict execution accuracy, which reports the fraction of queries that achieve a perfect match (F1 = 1.0) with the expected result table and the same row order, if relevant. As shown in Figure 7.5, the iterative approach reaches an execution accuracy of 21.4%, roughly double that of the Retriever–Rewriter (9.9%) and more than seven times higher than the Retriever-Reader (2.9%). This shows that the iterative process not only improves partial correctness but also increases the number of entirely correct query executions, validating that repeated reasoning and verification steps can resolve mismatches that single-step models miss.

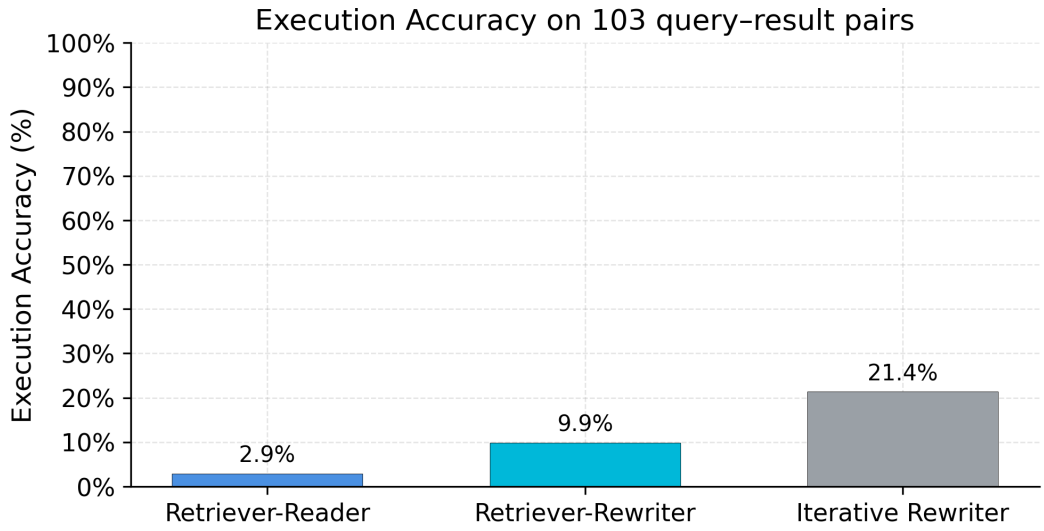

Figure 7.5: **Overall comparison of execution accuracy**. The illustration shows the execution accuracy of the baseline approaches (Retriever–Reader and Retriever–Rewriter) compared to the proposed Iterative Rewriter approach across all 103 query–result pairs.

Performance by Difficulty. Table 7.1 reports average F1 per query grouped by query difficulty. The results reveal that, regardless of difficulty, the Iterative Rewriter consistently outperforms the Retriever-Rewriter, which in turn outperforms the Retriever–Reader. At the same time, all approaches exhibit a clear drop in performance as query complexity increases, with the largest decline observed for hard queries involving multi-table joins and noisy data. The iterative approach shows the smallest degradation across difficulty levels, indicating stronger robustness to complex scenarios.

The Retriever-Reader performs worst on difficult queries because it must generate the entire output table directly from the limited context provided. Since we cannot always include all potentially relevant rows and columns within the model's context window, it is unlikely that the Retriever-Reader receives enough information to produce correct results.

Efficiency and Cost. Table 7.2 compares the average runtime and cost of each method. The Retriever–Rewriter baseline achieves the lowest latency and cost, completing in only 17.11s per query at a cost of $0.0091. In contrast, the Retriever-Reader baseline is more than twice as slow (44.99 s) and slightly more expensive ($0.0096). This difference arises because the Retriever-Reader requires larger table

Table 7.1: Average F1 per query grouped by query difficulty.

Approach	Easy	Medium	Hard
Retriever–Reader	0.32	0.31	0.16
Retriever-Rewriter	0.47	0.42	0.31
Iterative Rewriter	**0.68**	**0.67**	**0.55**

previews (20 rows instead of 10) to enable direct table generation, thereby significantly increasing the number of tokens per query. Moreover, reconstructing the entire result table from textual context is usually a more complex task than producing a rewritten SQL query. The Iterative Rewriter, while the most expensive and slowest approach (115.68 s, $0.0727), performs multiple refinement and retrieval–rewrite cycles. This additional computation enables it to achieve the highest final accuracy, highlighting the trade-off between runtime efficiency and execution quality.

Table 7.2: Average latency and cost per query for all methods.

Approach	Latency (s)	Cost (USD/query)
Retriever-Reader	44.99	0.0096
Retriever-Rewriter	17.11	0.0091
Iterative Rewriter	115.68	0.0727

Performance by Difficulty Type. A finer-grained view by difficulty type (Table 7.3), as introduced in Section 7.1, reveals how the different systems cope with specific user assumptions and noisy data. Across all categories, the Iterative Rewriter consistently outperforms both baselines. The largest improvements are observed for queries affected by Noise, Obsolete Tables, and Union queries, where iterative schema reasoning and self-checking enable the system to explore multiple table candidates and handle noisy inputs before execution. Smaller but still meaningful gains occur for Key Mismatch and Filter Mismatch, where value lookups (`SEARCH_VALUE`) and join path discovery improve execution accuracy.

Model Comparison for the Iterative Rewriter. Figure 7.6 compares three LLM backends used by the Iterative Rewriter. GPT-5 mini attains the highest accuracy (F1 0.65) at the cost of higher runtime (115.7s/query) and cost ($0.0727/query). GPT-4.1 mini offers a balanced trade-off (F1 0.57) with lower latency (52.8s) and cost ($0.0322). GPT-5 nano is the cheapest ($0.0157) but yields the lowest accuracy (F1 0.50) and the highest latency (255.4s). Note that the higher latency of GPT-5 nano cannot be attributed to more iterations, as it terminates on average after 3.05 iterations, comparable to GPT-4.1 mini, which completes after 2.94 iterations. The results highlight a clear trade-off among accuracy, cost, and latency

Table 7.3: Average F1 per query by error type.

Error Type	#Pairs	Retriever–Reader F1	Retriever–Rewriter F1	Iterative Rewriter F1
Missing Table	31	0.39	0.56	**0.64**
Name Mismatch	89	0.29	0.43	**0.64**
Obsolete Table	16	0.18	0.24	**0.59**
Noise	21	0.18	0.22	**0.64**
Union	10	0.19	0.20	**0.69**
Key Mismatch	20	0.28	0.47	**0.67**
Filter Mismatch	21	0.20	0.41	**0.68**

when selecting a model for iterative rewriting. GPT-5 mini is best for achieving the highest F1, GPT-4.1 mini offers the lowest latency, and GPT-5 nano provides the most cost-efficient option. Unless stated otherwise, results are reported with GPT-5 mini as the default.

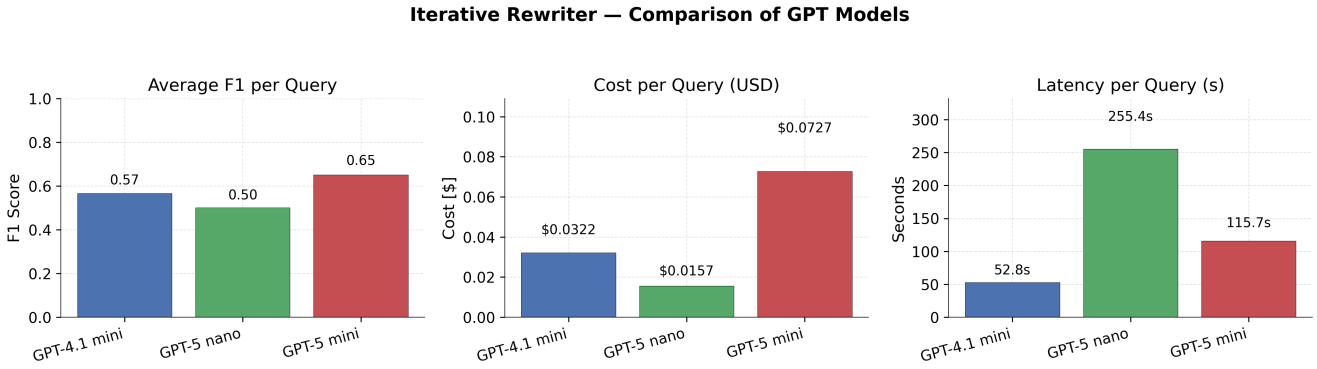

Figure 7.6: **Comparison of GPT model variants for the Iterative Rewriter**. The illustration shows the average F1, cost, and latency per query of the Iterative Rewriter using GPT-4.1 mini, GPT-5 nano, and GPT-5 mini.

7.4.2 Ablation Study

To better understand the contribution of each component in our iterative SQL rewriting framework, we conduct a detailed ablation study. Each variant disables one specific feature while keeping the rest of the pipeline unchanged. We evaluate the following five ablations:

- **No BUILD_CTE:** Disables the construction of auxiliary normalization subqueries, preventing the system from cleaning or unifying inconsistent data.

- **No SEARCH_TABLE:** Removes the ability to retrieve additional tables based on semantic similarity with more tailored specifications compared to the basic retrieval.
- **No UNION_SEARCH:** Prevents a dedicated search for unionable tables.
- **No SEARCH_VALUE:** Disables value lookups for correcting filter predicates or validating join keys.
- **No JOIN_GRAPH:** Takes away Table–Table relevance in retrieval and the `FIND_JOIN_PATH` action, leaving the system to infer joins from table previews without being able to specifically search for the tables/keys necessary to perform those joins.

Overall Results. Figure 7.7 summarizes the overall performance of each ablation variant. Removing any individual capability consistently reduces the F1 score compared to the complete approach (0.65). The largest degradations occur when disabling `BUILD_CTE` (0.58) or `SEARCH_TABLE` (0.58), followed by the `SEARCH_VALUE` ablation (0.59). Even less impactful modules like `JOIN_GRAPH` (0.62) and `UNION_SEARCH` (0.64) still contribute to robustness.

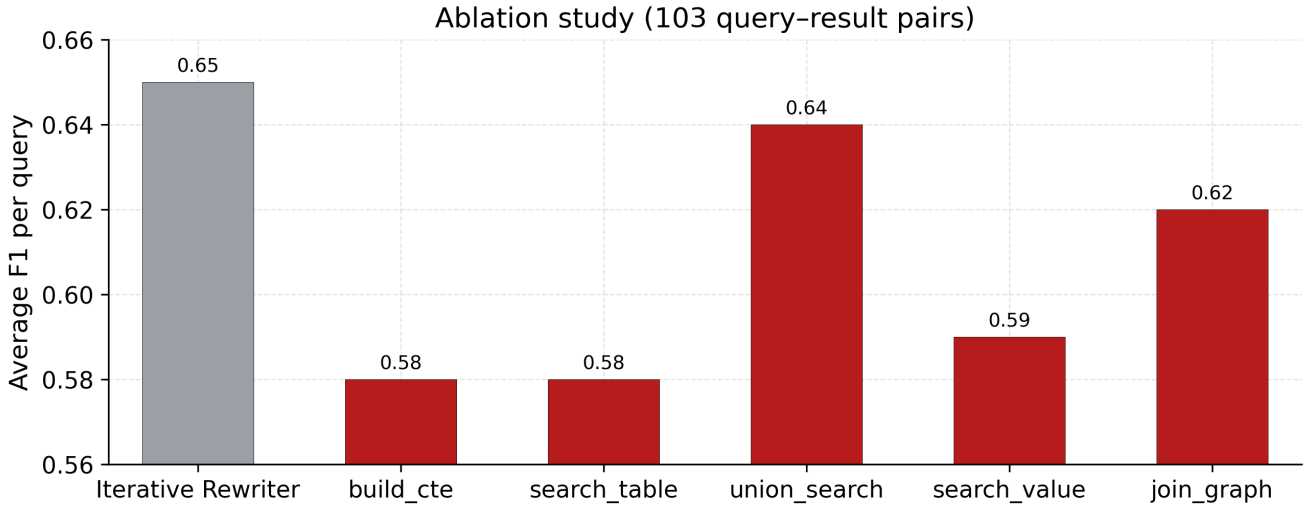

Figure 7.7: **Overall F1 comparison for ablation variants**. The illustration shows the average F1 scores per query of the proposed Iterative Rewriter compared to its ablated variants, each omitting a specific component, across all 103 query–result pairs.

Impact by Error Type. A breakdown by difficulty types (Figure 7.8) reveals how each module affects specific types of schema mismatches and errors. Overall, the complete system performs strongest across all categories, but the pattern of degradation differs depending on which capability is removed.

The most notable observations are:

- **No BUILD_CTE.** This ablation causes the largest drop on queries affected by Noise and Union errors (down to 0.44 F1 in both cases). Since the model can no longer normalize inconsistent formats (e.g., string casing, diacritics, or value type mismatches), filters, unions, and joins on noisy columns frequently fail.

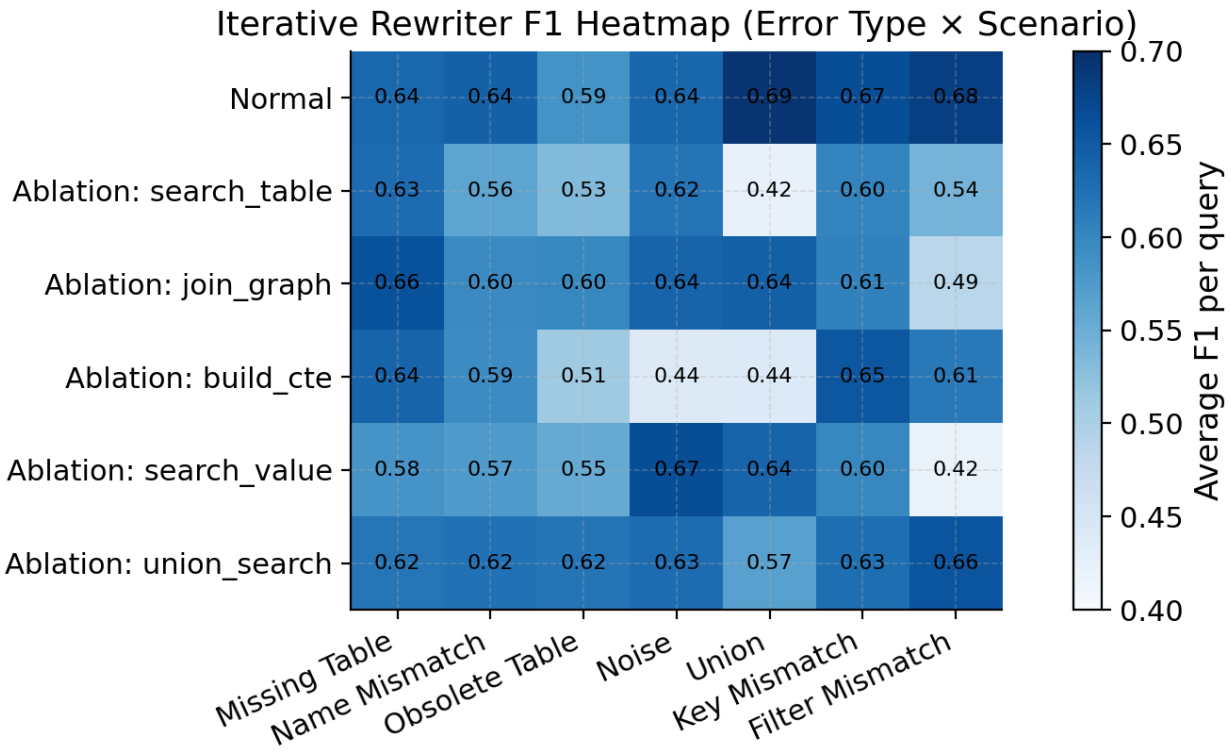

Figure 7.8: **Average F1 per query by error type for ablation variants**. The illustration shows a heatmap of average F1 scores per query across different error types and ablation variants, comparing how each system component contributes to handling specific query error scenarios.

- **No SEARCH_TABLE.** Disabling this action prevents the system from retrieving additional missing tables during iteration. While the overall impact is moderate for most error types, performance drops sharply for Union queries (0.69→0.42). Without `SEARCH_TABLE`, the model often fails to discover new base tables that would enable constructing a valid union, causing `UNION_SEARCH` to underperform as well. Since `UNION_SEARCH` relies on an existing base table to identify compatible union partners, missing these initial grounding tables limits its effectiveness. Smaller declines for Name Mismatch (0.64→0.56) and Obsolete Table (0.59→0.53) further indicate reduced flexibility when substituting semantically related tables.
- **No JOIN_GRAPH.** Removing the Join Graph eliminates precomputed joinability cues, forcing the system to infer relationships solely from schema previews. The Join Graph is particularly effective when two tables can be joined directly, as it reliably identifies the correct key of a join. However, when intermediate tables are required to connect distant relations, the search space becomes highly ambiguous. In such cases, incorrect intermediate tables often outweigh the correct ones, reducing precision and making multi-hop joins less reliable. This explains why Missing Table queries are slightly stronger in the ablation, while Key Mismatch (0.67→0.61) and Filter Mismatch (0.68→0.49) queries degrade more strongly, since they often only rely on accurately resolving direct joins to preserve key and filter semantics.
- **No UNION_SEARCH.** Removing this action has only a minor impact in the current setting, as the

small size of the data lake allows SEARCH_TABLE and basic retrieval to cover most unionable sources. However, UNION_SEARCH becomes increasingly important as the data lake scales, where more specialized searches are needed to identify unionable tables while respecting type compatibility and structural constraints.

- **No SEARCH_VALUE.** Disabling this action results in a significant drop in Filter Mismatch performance (0.68→0.42). Without SEARCH_VALUE, the system can no longer perform targeted lookups to verify or correct filter predicates, forcing it to rely solely on column-name similarity and the few preview values stored in the scratchpad.

7.4.3 System Behavior Analysis

This subsection analyzes how the iterative system behaves during execution: how quickly it discovers the required tables, how the scratchpad evolves, and which actions are most frequently used.

Required-Table Discovery and Coverage. Table 7.4 summarizes discovery statistics over all runs. On average, the system terminates after 2.9 iterations, and in completed runs, it reaches full table coverage by iteration 1.5. Overall, the system discovers on average 77% of the required tables.

Table 7.4: Required tables discovery summary.

Metric	Value
Success fraction (all required tables found)	0.670
Average termination iteration	2.913
Average iteration to first full coverage (completed cases only[1])	1.522
Average final coverage of required tables	0.772

Scratchpad Growth and Eviction. Table 7.5 report how the scratchpad evolves. Across runs, sections remain sparingly populated (automatic eviction due to section-size limit was triggered in only 13/103 cases), and the LLM never chose the explicit EVICT_TABLE action (by design, it is prompted to use evictions defensively). Given our evaluation lake size (83 tables), constrained context is rarely a bottleneck. We expect eviction strategies to become more important at larger scales.

Action Usage and Diagnostics. Table 7.6 summarizes how often the self-checker invoked each action (note that the same action may be selected multiple times in a single iteration with different parameters). SEARCH_VALUE is the most frequent action (1.068 per run). Crucially, it is not only used to repair filter predicates, it is also employed to validate prospective join keys and probe value overlap between candidate columns before committing to a join. For example, in a case where a join

[1]Only includes cases where all required tables were successfully retrieved.

Table 7.5: Scratchpad section counts (last iteration).

Section	Total	Avg per run
Base Table Previews	517	5.019
Join Previews	92	0.893
CTE Previews	39	0.379
Value Search Previews	95	0.922
UDFs	97	0.942

on `full_name` produced an empty result, the self-checker issued a sequence of `SEARCH_VALUE` operations to confirm whether representative person names co-occurred in both tables and, if not, whether alternative identifier columns should be considered or normalized via `BUILD_CTE`.

Table 7.6: Action usage across all runs.

Action	Total	Avg per run
BUILD_CTE	68	0.660
EVICT_TABLE	0	0.000
UNION_SEARCH	17	0.165
SEARCH_TABLE	67	0.650
SEARCH_VALUE	110	1.068

On (Over)Use of `BUILD_CTE`. We observe `BUILD_CTE` in at least one iteration for 43 examples, despite being truly required in only 21. These false-positive cleanups typically arise when the model suspects value-format issues (e.g., name variants or partially corrupted fields) and inserts unnecessary normalization steps. Although such steps can slow execution without immediate accuracy gains or even slightly harm output quality due to false-positive cleanups, they substantially increase robustness in noisy scenarios. Our ablation in Section 7.4.2 confirms that removing `BUILD_CTE` is among the most damaging changes overall.

Candidate Selection Quality. The self-checker's ranking of candidate rewrites works well in practice: the selected final candidate is the best (highest F1 among candidates) with 0.835 accuracy. This suggests that our execution feedback is informative enough for robust model-side selection.

8 Discussion

This section interprets the experimental findings and places them in context with the research objectives outlined earlier. We discuss how the iterative SQL rewriting framework improves upon single-step baselines, analyze the implications of the ablation results, and reflect on observed system behaviors. Finally, we highlight key limitations and provide a critical assessment of the framework's performance and robustness.

Main Findings. The results demonstrate that iterative SQL rewriting substantially outperforms single-step baselines by progressively correcting schema mismatches and handling noisy data. The feedback loop decomposes complex queries into smaller reasoning steps, enabling the system to refine intermediate hypotheses and correct earlier assumptions. The largest improvements appear for hard queries, particularly those involving incorrect data decompositions and noisy input. These findings confirm that iterative refinement offers a more reliable strategy for executing queries over heterogeneous and loosely structured data lakes than single-pass reasoning.

Iterative Reasoning. While the iterative process increases accuracy, it also incurs higher costs and latency than the baselines. Depending on the target application and query difficulty, the single-step Retriever–Rewriter may remain a practical choice when efficiency outweighs marginal gains in accuracy. In contrast, the Retriever–Reader consistently performs worse in all dimensions, confirming that directly predicting the result table is less effective than reasoning symbolically through SQL rewriting.

The results further support the decision to make the framework iterative. In practice, relevant tables are rarely retrieved all at once, especially for complex queries involving multiple joins or schema inconsistencies. The system typically requires multiple reasoning steps to achieve full table coverage, illustrating that information discovery in data lakes is inherently incremental. This highlights a fundamental limitation of single-pass approaches, which unrealistically assume that all relevant evidence can be retrieved in a single retrieval cycle.

Evaluation Considerations. The data lake contains substantial overlap across tables, which allows the system to produce partially correct results even when not all required tables are discovered. This redundancy reflects a common real-world challenge: multiple SQL rewrites can satisfy the user's intent while returning slightly different, yet semantically valid, results. As a result, strict execution accuracy tends to underestimate actual performance, as multiple valid query formulations can yield slightly different but equally meaningful results.

Model Comparison. Changing the underlying LLM backend has a measurable effect on performance characteristics. GPT-5 mini is best for achieving the highest F1, GPT-4.1 mini offers the lowest latency, and GPT-5 nano provides the most cost-efficient option. Depending on the use case, different models may be preferable, underscoring the need for model selection to align with the desired trade-off between accuracy, cost, and runtime efficiency.

Component Behavior and Noise Handling. One of the key strengths of the proposed approach lies in its specialized data lake actions, which enable the system to adaptively retrieve and refine evidence aligned with specific information needs. The ablation results show that every action contributes to the overall system performance, although some actions reveal clear limitations when analyzed by error type.

While the Join Graph improves direct join reasoning, it still struggles with multi-hop join discovery through intermediate tables. This task remains inherently ambiguous due to overlapping schemas and partially correlated value spaces. As join path length increases, the search space grows combinatorially, and minor local errors can propagate through the reasoning chain. Similarly, while `BUILD_CTE` significantly enhances robustness against noisy data, data cleaning remains a challenging open problem in large, heterogeneous lakes. For example, the current Join Graph assumes relatively clean data, which simplifies joinability estimation but limits robustness in real-world settings. At present, noise detection is intentionally conservative to minimize false positives, focusing only on clear and recurring error patterns. While this design reduces unnecessary cleaning, it may overlook subtle inconsistencies such as encoding artifacts, type mismatches, or partial normalization errors.

Scalability and System Design. The scalability of the framework depends on both the size of the data lake and the complexity of its relational structure. The Join Graph construction currently scales quadratically with the number of tables. Join path inference forms a second bottleneck: identifying correct paths through multiple intermediate tables is computationally expensive and increasingly uncertain as schema complexity grows. Retrieval precision also deteriorates at scale, underscoring the importance of targeted actions such as `SEARCH_TABLE` and `UNION_SEARCH`, which narrow the search space to contextually relevant regions. Finally, efficient context management and table eviction, while largely unnecessary in small experiments, become critical for preventing context overflow and maintaining consistent reasoning depth as the number of tables increases.

Query Limitations. Despite its progress, the system continues to face challenges with complex queries that combine multiple error types, such as incomplete joins, heavy noise, redundant tables, or deeply nested subqueries. In such cases, the iterative loop may fail to retrieve the proper evidence, terminate prematurely, or misprioritize actions, leading to incomplete rewrites or suboptimal results.

9 Conclusion

This final chapter concludes with a summary of our work and an outlook on promising future work.

9.1 Summary

Modern organizations increasingly rely on data lakes to store vast amounts of heterogeneous and loosely structured data. While this flexibility simplifies large-scale data integration, it complicates querying the data: unlike traditional relational databases, data lakes lack schema information such as explicit table and column names and foreign-key relationships. Additionally, they often contain noisy or inconsistent values, ambiguous naming conventions, and partially overlapping tables. As a result, querying a data lake involves making assumptions about its underlying structure and content, which can lead to various errors.

This thesis addressed the challenge of enabling structured query execution over disorganized data lakes through an iterative SQL rewriting framework powered by LLMs. The proposed system iteratively transforms user-written SQL queries into executable queries that operate correctly on the actual data lake and fulfill the user's intent. To achieve this, the framework integrates an iterative feedback loop in which the system repeatedly analyzes execution results and performs tailored retrieval actions, such as searching for missing tables, discovering join paths, or cleaning noisy data, to progressively refine the query and align it with the data lake's structure.

An extensive evaluation over a curated benchmark of 103 query–result pairs derived from the WikiDBs corpus showed that iterative rewriting substantially improved query execution accuracy compared to strong single-step baselines. Ablation studies further revealed which components contributed most strongly to the performance gains, underscoring the importance of tailored retrieval mechanisms and data cleaning. Overall, this work presented the first systematic evaluation of LLM-driven query rewriting over noisy, schema-free data lakes and demonstrated the potential of iterative reasoning to bridge the gap between user intent and real-world data heterogeneity.

9.2 Future Work

While the proposed framework achieves substantial improvements, several challenges remain. The system's ability to infer complex multi-hop join paths and handle queries that combine multiple error types, such as missing joins, redundant tables, or nested subqueries, remains limited. Addressing such cases likely requires deeper structural reasoning and more explicit control over multi-hop search

and feedback strategies. Future work should therefore explore improved action selection mechanisms, hierarchical planning approaches, and uncertainty estimation to better handle these compound scenarios.

From a scalability perspective, Join Graph construction currently scales quadratically with the number of tables, suggesting that incremental or approximate graph-building strategies will be essential for larger deployments. Similarly, retrieval precision deteriorates under scale, indicating the need for more effective mechanisms to narrow down the search space and focus reasoning on the most relevant regions of the data lake as it grows in size and complexity.

Noise handling remains another open challenge. The current system employs conservative detection to avoid false positives, limiting its ability to address subtle inconsistencies, such as encoding errors or partial normalization. Future work should focus on detecting more nuanced forms of noise while preserving precision and avoiding over-correction, for instance, by combining lightweight statistical heuristics with LLM-based pattern recognition.

Overall, this thesis contributes to bridging the gap between structured relational databases and unstructured data lakes by demonstrating how iterative reasoning can transform noisy, weakly structured collections into queryable resources. By integrating symbolic database principles with LLM-guided adaptability, such systems move toward a more general paradigm of self-correcting and data-aware reasoning, an essential step toward the next generation of intelligent data management systems.

Acknowledgements

At this point, I would like to thank all those who supported me in writing this thesis. In particular, I would like to thank:

- My supervisor Jan-Micha Bodensohn for his exceptional support. Our frequent personal meetings provided valuable insights, inspiring ideas, and constructive feedback that opened my mind to many new questions and perspectives throughout the thesis.
- Professor Dr. Carsten Binnig for his continued guidance and support throughout the thesis.
- My friends and fellow students at Technische Universität Darmstadt for the motivating discussions.
- My family for their continuous support throughout my studies.

List of Tables

List of Figures

References

Biswal, Asim et al. (Aug. 27, 2024). *Text2SQL Is Not Enough: Unifying AI and Databases with TAG*. DOI: 10.48550/arXiv.2408.14717. arXiv: 2408.14717 [cs]. URL: http://arxiv.org/abs/2408.14717. Pre-published.

Chen, Peter Baile et al. (Jan. 9, 2025). *Is Table Retrieval a Solved Problem? Exploring Join-Aware Multi-Table Retrieval*. DOI: 10.48550/arXiv.2404.09889. arXiv: 2404.09889 [cs]. URL: http://arxiv.org/abs/2404.09889. Pre-published.

Hai, Rihan et al. (June 17, 2021). *Data Lakes: A Survey of Functions and Systems*. arXiv.org. DOI: 10.1109/TKDE.2023.3270101. URL: https://arxiv.org/abs/2106.09592v2.

Herzig, Jonathan et al. (June 9, 2021). *Open Domain Question Answering over Tables via Dense Retrieval*. DOI: 10.48550/arXiv.2103.12011. arXiv: 2103.12011 [cs]. URL: http://arxiv.org/abs/2103.12011. Pre-published.

Lei, Fangyu et al. (Mar. 17, 2025). *Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows*. DOI: 10.48550/arXiv.2411.07763. arXiv: 2411.07763 [cs]. URL: http://arxiv.org/abs/2411.07763. Pre-published.

Li, Jinyang et al. (Nov. 15, 2023). *Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs*. DOI: 10.48550/arXiv.2305.03111. arXiv: 2305.03111 [cs]. URL: http://arxiv.org/abs/2305.03111. Pre-published.

Reimers, Nils and Iryna Gurevych (Aug. 27, 2019). *Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks*. DOI: 10.48550/arXiv.1908.10084. arXiv: 1908.10084 [cs]. URL: http://arxiv.org/abs/1908.10084. Pre-published.

Vogel, Liane et al. (Nov. 13, 2024). "WikiDBs: A Large-Scale Corpus Of Relational Databases From Wikidata". In: The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track. URL: https://openreview.net/forum?id=abXaOcvujs#discussion.

Wu, Jian et al. (Oct. 4, 2024). "MMQA: Evaluating LLMs with Multi-Table Multi-Hop Complex Questions". In: The Thirteenth International Conference on Learning Representations. URL: https://openreview.net/forum?id=GGlpykXDCa.

Yen, Jin Y. and Jin Y. YENt (2007). "Finding the K Shortest Loopless Paths in a Network". In: URL: https://www.semanticscholar.org/paper/Finding-the-K-Shortest-Loopless-Paths-in-a-Network-Yen-YENt/aa6a64afc25f48ad44e510d0055405836c8cc325.

Yu, Tao et al. (Feb. 2, 2019). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. DOI: 10.48550/arXiv.1809.08887. arXiv: 1809.08887 [cs]. URL: http://arxiv.org/abs/1809.08887. Pre-published.

Zhang, Xuanliang et al. (Sept. 18, 2024). *MURRE: Multi-Hop Table Retrieval with Removal for Open-Domain Text-to-SQL*. DOI: 10.48550/arXiv.2402.10666. arXiv: 2402.10666 [cs]. URL: http://arxiv.org/abs/2402.10666. Pre-published.