Master thesis
Computer Science

Radboud University

Query-Log-Informed Schema Descriptions and their Impact on Text-to-SQL

Author:
Evgeniia Egorova
s1122855

Daily supervisor:
Till Döhmen
till@motherduck.com

First supervisor/assessor:
Prof. Dr. Hannes Mühleisen
hannes.muehleisen@ru.nl

Second assessor:
Dr. Madelon Hulsebos
madelon@cwi.nl

August 22, 2025

Abstract

This thesis addresses the problem of sparse and low-quality documentation for database schema elements, a common issue in industry-sized data warehouses that complicates interactions with the data for both human users and AI agents. To mitigate this problem, we suggest introducing informative textual descriptions for schema elements which are automatically generated by leveraging historical query logs. These can provide necessary guidance for users and serve as a semantic context for AI-powered applications.

We explore several ways to produce such descriptions from query history and evaluate the utility of these generated descriptions as contextual enhancements for LLMs in Text-to-SQL tasks. Our experiments, conducted on both the BIRD benchmark and a custom real-world dataset, demonstrate that query-log-informed descriptions are beneficial for SQL generation performance, particularly in scenarios with ambiguous identifiers and repetitive query patterns. While the impact on BIRD ranged from 1.5% to 3%, the production dataset saw improvements of up to 10%, highlighting the practical potential of our approach in industrial settings.

Contents

1 Introduction **3**

1.1 Motivation . . . 3
1.2 Goal and Research Questions . . . 4
1.3 Contributions . . . 4
1.4 Thesis Structure . . . 5

2 Background **6**

2.1 SQL . . . 6
2.2 DuckDB and MotherDuck . . . 7
2.3 Text-to-SQL Objective . . . 7
2.4 Large Language Models . . . 8
2.5 LLMs and SQL . . . 9

3 Related Work **11**

3.1 Semantic Metadata for Table Understanding . . . 11
3.1.1 Pre-trained Language Models . . . 11
3.1.2 Large Language Models . . . 12
3.2 Semantic Metadata for Text-to-SQL . . . 13
3.3 Generating Semantic Metadata . . . 16
3.4 Modern Data Catalogs . . . 17

4 Methodology **19**

4.1 Query logs . . . 19
4.1.1 Query Annotation . . . 20
4.1.2 Query Pattern Mining . . . 21
4.2 Baseline Methods: Schema & Column profiling . . . 30
4.3 Generating Descriptions . . . 30

5 Experimental setup **33**

5.1 Data . . . 33

5.1.1 BIRD . . . 33
5.1.2 MDW-AMBIG . . . 34
5.1.3 BIRD and MDW-AMBIG comparison . . . 36
5.2 Models . . . 40
5.3 Evaluation . . . 40
5.4 Text-to-SQL Setup . . . 41
5.4.1 Schema . . . 41
5.4.2 Samples & Descriptions . . . 42
5.4.3 Other . . . 43

6 Results **44**
6.1 BIRD results . . . 44
6.2 MDW-AMBIG . . . 47

7 Discussion **51**
7.1 Overall performance: How useful are query-log-informed descriptions for LLMs in SQL generation? . . . 51
7.2 Comparing to the official BIRD leaderboard . . . 52
7.3 Benefits and Limitations of query-logs-informed descriptions . . . 53
7.3.1 Advantages . . . 53
7.3.2 Disadvantages . . . 53
7.4 Future Work . . . 55

8 Conclusions **57**

A Generation Prompts **68**
A.0.1 System Prompt . . . 68
A.0.2 Schema, stats and Query Annotation Description Prompt for Columns 69
A.0.3 Schema, stats and Query Annotation for Description Prompt Tables 70
A.0.4 Query Patterns Description Prompt . . . 71
A.0.5 Query Patterns Table Description Prompt . . . 72
A.1 SQL Generation Prompts . . . 72
A.1.1 System Prompt . . . 73
A.1.2 User Prompt Template for SQL Generation . . . 73
A.2 Cost Analysis for Description Regeneration . . . 73

Chapter 1

Introduction

1.1 Motivation

In practice, data warehouses continuously grow in both size and complexity, making them progressively more difficult for data engineers to maintain and for analysts and data scientists to use effectively. Tasks such as onboarding new users or locating and exploring unfamiliar data can become highly time-consuming and tedious. In addition to size, one of key issues also lies in the inherent ambiguity of the schema elements which often include obscure table or column names, uninterpretable without prior, often tribal, knowledge. This creates three primary challenges in managing data: data discovery, where users struggle to find the right data; data interpretability, where they cannot understand what the data represents; and data usage, where knowing how to properly query and interact with the data becomes a non-trivial task.

These challenges affect not only human users but also artificial intelligence (AI) systems. In the current era when agentic frameworks are rapidly gaining popularity, it is important to develop efficient workflows for agents to interact with data storage systems. One step in this direction is the introduction of Model Context Protocol which streamlines integration between AI agents and external tools, including data warehouses. However, the fundamental issues with understanding and navigating data still persist: Large Language Models (LLMs), which are a backbone of agentic systems, struggle to deal with enormous schemas and similar or confusing table/column names. Despite their advanced capabilities, LLMs require rich context to perform data-related tasks effectively.

One intuitive solution is to provide documentation about the meaning of schema elements along with usage guidelines. Unfortunately, despite its usefulness, documentation is often neglected in industrial settings as it is difficult to create and maintain. In this study, we aim to address this challenge by automating the documentation process.

A natural choice for documentation format is *free text*: it is flexible enough to capture

diverse kinds of information yet is consumable by both human users and LLMs, thus, has a potential to be adopted in a variety of applications. However, automatically producing insightful documentation requires a rich source of contextual information. In context of data warehouses, one promising candidate is a *collection of historical queries*: they encode how users interact with data, implicitly revealing semantics about the underlying schema.

1.2 Goal and Research Questions

In this study, we aim to automatically generate documentation for schema elements: tables and columns. We opt to structure this documentation in free-text form due to its flexibility and potential for adoption. To produce insightful descriptions without user input, we draw on query history - a source that is usually available in enterprise settings yet densely packed with contextual information. We investigate methods for extracting and structuring information from query logs and evaluate the utility of the resulting descriptions in Text-to-SQL task, one of the potential applications.

In summary, this thesis addresses the following research questions:

- **RQ1:** How can informative descriptions for schema elements be automatically synthesized from existing query history?
- **RQ2:** To what extent are such descriptions beneficial for LLMs in the context of SQL generation?

1.3 Contributions

In pursuit of these research questions, this thesis makes the following key contributions:

- the design and implementation of a methodology for generating schema descriptions from information extracted from query logs;
- an empirical evaluation demonstrating the effect of incorporating such descriptions on Text-to-SQL performance.

In addition, we provide several side contributions:

- the development of a SQL query parser based on Abstract Syntax Trees (AST), capable of extracting all table and column references as well as expressions used in queries;
- the non-public dataset of real-world queries executed by users against a production-level internal company database, along with an analysis of its discrepancies compared to commonly used Text-to-SQL benchmarks.

1.4 Thesis Structure

The rest of the thesis is organized as follows. In Chapter 2, we discuss the tools and concepts relevant to our study, such as LLMs, SQL generation, and DuckDB. Chapter 3 reflects on how documentation is used by AI in various data management and table-related tasks, with a particular focus on Text-to-SQL methods. It also reviews existing approaches for generating schema element descriptions. The details of how we decompose query history and translate it into text form are presented in Chapter 4. Chapter 5 covers the implementation of our Text-to-SQL experiments and evaluation framework. The results of applying different generated descriptions to Text-to-SQL are discussed in Chapter 6, while Chapter 7 provides a broader perspective on the proposed methodology and potential directions for future work. Finally, Chapter 8 concludes the study by summarizing the key outcomes of our work.

Chapter 2

Background

This section introduces important concepts related to this study and establishes terminology that will be used throughout the paper. We discuss the technologies on top of which this study was built, such as DuckDB, MotherDuck, and Large Language Models (LLMs).

2.1 SQL

Structured Query Language (SQL) is a standard language for storing, managing, and retrieving data in relational databases. It provides commands to manage the stored data with creating tables, inserting, updating and removing records. SQL is also used for analytical purposes to filter, aggregate, and join data, making it possible to answer complex questions with data.

SQL belongs to a declarative type of language: it describes *what* data to retrieve rather than *how* to retrieve it, the database engine handles execution details such as indexing and query optimization. This makes it accessible to users without deep knowledge of low-level data management while still powerful enough for large-scale analytics.

SQL is supported by Database Management Systems, each with its own *dialect* - variations in syntax, functions, and features. Widely adopted systems include PostgreSQL and MySQL which are used in everything from small web applications to large enterprise infrastructures. SQLite is another popular embeddable system, widely used due to its small footprint and zero-configuration design. Newer analytical engines like DuckDB, BigQuery, and Snowflake are growing in adoption for high-performance analytics, though they have not yet reached the popularity of the more established systems.

2.2 DuckDB and MotherDuck

In this thesis, we mainly work with DuckDB databases and queries. DuckDB is a modern, in-process analytical database management system designed for fast execution of analytical queries on large datasets. DuckDB is lightweight, embeddable, and requires no external server, making it especially suitable for integration into data science workflows. Its efficient storage and processing of tabular data enables DuckDB to do analytics directly on local machines with performance comparable to larger distributed systems.

DuckDB implements its own SQL dialect, largely following the PostgreSQL standard. Its SQL syntax is designed to be expressive yet lightweight, which makes it easy to adopt by users already familiar with mainstream relational systems. Importantly to this study, it supports *Common Table Expressions (CTEs)*, a temporary named result set that can be referenced within the same query. CTEs largely improve readability in comparison to subqueries, which is why they are often used in complex analytical queries. In practice, analytical queries tend to be large and diverse, often chaining together multiple operations and intermediate results.

This work has been done in collaboration with MotherDuck, a cloud-based data warehouse built around DuckDB that extends its capabilities to collaborative and large-scale use cases. It enables analytical workloads to be executed both locally and in the cloud, and provides users with a web interface enriched by AI-powered features. These include an automatic syntax error fixer, triggered whenever a query fails to execute, and a SQL assistant that can generate or refine queries based on user instructions. For these features, user experience is a critical aspect; thus, in addition to correctness, there are also strict latency constraints. Automatic documentation generation is currently one of the investigated areas, with results that can be applied in a variety of other features. This is partially why in this study we paid more attention to the efficiency and versatility factors when designing the approach proposed in this study.

2.3 Text-to-SQL Objective

Given the prominence and widespread adoption of relational data, numerous table-related objectives have naturally emerged in the field of Artificial Intelligence. [FXT+24] distinguishes 3 main directions:

- **Tabular Prediction** – using tabular data for classification or regression in a traditional Machine Learning sense;
- **Data generation** – synthesizing high-quality relational data, for instance, as augmentations or to replace `NULL` values;

- **Table Understanding** – interpreting semantics and structure of tabular data for downstream applications, e.g., answering natural language questions about the data or resolving semantic column types.

Text-to-SQL, the primary focus of this study, falls under Table understanding. It is often referred to as "semantic parsing" [YNYR20] and is naturally a subkind of it. The main objective of Text-to-SQL is to translate a natural language question into a SQL query, which can then be executed against a database to answer the original question.

Text-to-SQL Translation Example

Natural Language Question:
"What is the average salary of employees in the Marketing department?"

Generated SQL Query:
```
SELECT AVG(e.salary)
FROM employees e
JOIN departments d ON e.department = d.id
WHERE d.name = 'Marketing';
```

Figure 2.1: Example of Text-to-SQL translation process

In line with the evolution of AI as a field, text-to-SQL methods have progressed from rule-based approaches to neural networks-based architectures (mainly sequence-to-sequence and graph- models) and more recently pre-trained language models [LSL+25], [LLC+24]. Nowadays text-to-SQL is predominantly tackled using Large Language Models which offer the advantages of not requiring pre-training and smooth adaptation to many domains or SQL dialects.

2.4 Large Language Models

Large Language Models are advanced neural networks trained on large collections of textual data to predict and generate coherent sequences of words. They are most commonly based on the transformer architecture [VSP+23], which relies on self-attention mechanisms to capture long-range dependencies in text. This architecture has proven highly scalable, allowing to improve the performance significantly with increases in model size, training data, and compute power.

Users interact with LLMs via textual instructions (prompts), and the output of models is also in natural language. The models posses strong general-purpose reasoning capabilities acquired through training on large-scale datasets, which enable them to perform a wide range of tasks effectively without fine-tuning for specific objectives. The combination of versatility and inherent intelligence has driven rapid adoption of LLMs across numerous industries and domains.

Since the release of the first-of-its-kind ChatGPT (GPT-3.5) by OpenAI in 2022, the LLM ecosystem has expanded significantly. Proprietary models such as OpenAI's GPT series, Anthropic's Claude, and Google's Gemini, accessible through paid services, co-exist with open-source alternatives like Meta's LLaMA, Mistral, and Alibaba's Qwen. Open-source LLMs have been reported to be generally less advanced than their commercial counterparts in many applications. However, their accessibility and the ability to fine-tune them for specific domains make them highly attractive for experimentation and customization.

2.5 LLMs and SQL

As with vast most sequence-to-sequence tasks, LLMs have proven to perform extremely well at SQL generation. A typical modern Text-to-SQL pipeline with LLMs is illustrated in Figure 2.2. The user provides a natural language question, which is incorporated into the prompt with contextual information about the underlying database. This context is usually presented by the database schema and a few content samples, both of which have proven to be crucial components of any Text-to-SQL prompt. The schema helps the model ground its predictions in the correct identifiers instead of hallucinating non-existent ones. Content samples, in turn, familiarize the model with the actual data and, importantly, formatting of values. It helps with data type conversions and prevents filtering mismatches: for instance, filtering on uppercase strings when the column values are lowercase. A big advantage of both schema and samples is that these elements are generally available [1] and straightforward to retrieve.

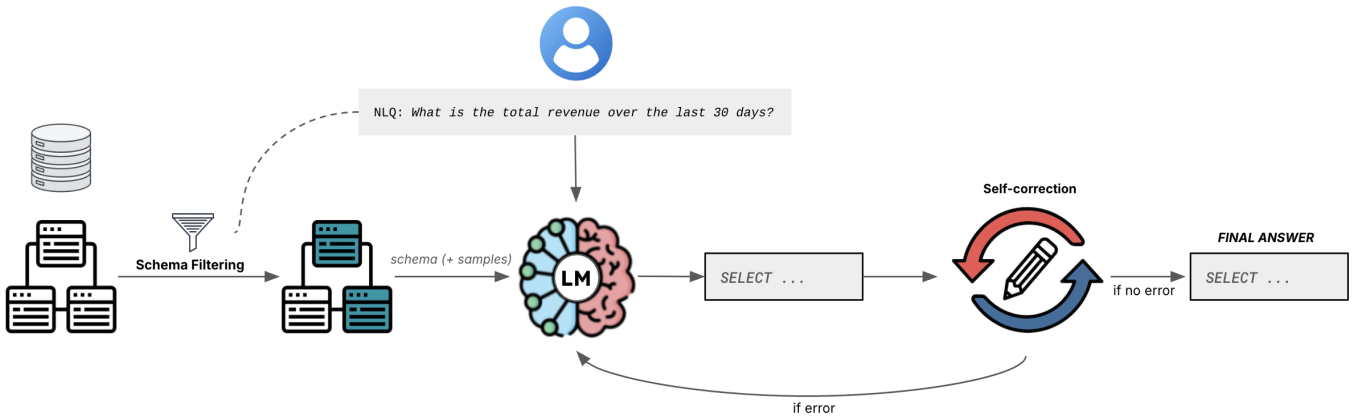

Figure 2.2: Traditional Text-to-SQL pipeline with LLM

However, including the full schema in the prompt can be problematic: usually not all

[1] With the exception of cases where privacy concerns restrict sharing data with proprietary models

tables and columns are used in the ground truth queries, resulting excessive information that "pollutes" the context. It can become especially challenging with very schemas. That is why usually only a subset of columns and tables, which are expected to be relevant to the user's question, is included in the prompt. This procedure is referred as *schema filtering* and is highly recall-sensitive: if even a single necessary element is omitted, SQL generation is basically doomed. While some argue that modern LLMs are capable enough to process longer context and so schema filtering should not be prioritized [**?**], there is still a performance gain if it is done effectively. In industry-scale databases its impact is likely to be more pronounced, and in certain cases schema filtering is not just beneficial but unavoidable, e.g. when the schema is so large that it does not fit in the model's context window.

Another module commonly used in Text-to-SQL systems is *self-correction.* After a query is generated, it is executed against the database; if execution fails, the prediction is for sure incorrect. In these cases, the resulting error message together with the previous prediction are added to the context, and the LLM is invoked again. Error messages in DuckDB are rather descriptive and can provide additional guidance for refining the prediction. This procedure can be repeated multiple times until the predicted query becomes executable and stands a chance to be correct.

Chapter 3

Related Work

Many systems working with tabular data utilize semantic table metadata. It can include column and table names (headers), their descriptions and other available documentation. This section describes how such metadata has been leveraged for table-related and data management tasks. We address Text-to-SQL systems in detail as this is the primary application we are going to evaluate our descriptions against. We also discuss some of the proposed approaches to generate and evaluate metadata, and schema descriptions in particular.

3.1 Semantic Metadata for Table Understanding

3.1.1 Pre-trained Language Models

The breakthrough of the Transformers architecture [VSP+17] introduced a paradigm of contextual data representations and transfer learning. The first step of a traditional pipeline includes obtaining vector representations of data from Pre-trained Language Models (PLM) which are subsequently used as an input to train task-specific classifiers. There are multiple studies which adapted this approach to tabular data [YGU+22], [YNYR21], [YNYR20], [DSL+20], [HNM+20], [LYKK22].

To enrich the semantic properties of produced embeddings, some systems opted to utilize metadata at the pre-training stage. For instance, [DSL+20] incorporates table captions and headers into the embeddings alongside the table content. During the pre-training, a model learns to restore masked tokens of the metadata in a traditional Masked Language Modeling manner, introduced in [DCLT18], while also optimizing to recover the entity cells as a part of Masked Entity Recovery objective. These embeddings are then used as inputs for classifiers finetuned for different Table Understanding or Table Augmentation tasks. Similar approach with recovering headers and data types of masked columns is utilized in [YNYR20] which is designed for semantic parsing tasks. [WSL+21]

makes use of a page topic which is appended to a table as a separate column and then is included in inter- and intra-table aggregations to form latent embeddings for cells, rows and columns. [MW23] is another approach leveraging metadata for the pre-training. The authors explore a contrastive learning framework applied to the tabular data domain and introduce methods to integrate the headers, captions and topics into the learning pipeline to expand the scope of positive and negative anchors. The experimentation results demonstrate that incorporating semantic metadata during the pre-training improves the quality of representations with an increasing performance on downstream tasks (Column Type Annotation and Relation Extraction).

Many approaches, however, opt to rely solely on table content as metadata is often unavailable, making heavy dependence on metadata a limitation. It is also challenging to incorporate the metadata into the pretraining. The majority of the sequence-to-sequence frameworks for Text-to-SQL encode only natural language questions, schema, and/or data values. For example, while [CYXH21] only uses graph representations of a question and schema, [LZLC23] experiments with adding relevant database content, which has been shown to be beneficial. [LSX20] and [WSL+20] also include information about primary-foreign key relationships in their encoding strategies.

3.1.2 Large Language Models

With the recent advancements of generative AI, Large Language Models (which are naturally a subset of Pretrained LMs but are often referred as a separate class of models capable of advanced natural language understanding and in-context learning [LLC+24]) became a common design choice for table reasoning systems. In comparison to PLMs, LLMs mostly work out-of-the-box and do not require specialized pretraining to adapt to new tasks and domains. This adaptation occurs as a part of so-called in-context learning when a model performs a new task given a natural language instruction as a prompt without updating the model's weights [FXT+24]. It can be done in a completely zero-shot setup; however, providing demonstrations and external context has proven to enhance the generation quality and avoid hallucinations and outdated knowledge [GXG+24], [BMR+20].

As mentioned in the background 2.5, it is essential to provide LLMs with informative context. When working with LLMs for table understanding, one of the most important parts of the prompt is the schema of the table. The majority of the frameworks also include samples of table content as examples. However, due to obscure identifiers and hard-to-interpret database values, which is a common scenario in real-world data warehouses, it is often found beneficial to further extend the context with additional metadata.

[SZZ+23] explores the use of metadata for table question answering and fact verification and experiments with including some table-related characteristics like Dimension/Measure (whether the column contains categorical values often used for grouping and filtering or

quantified and numeric values), Semantic Field Type, Statistics Features, etc, as well as external domain knowledge such as relevant web pages or term explanations. In the result different metadata properties appear to be useful for different datasets: for instance, providing Term Explanations as the context increases the scores on "datasets requiring complex text understanding and generation" but not "datasets involving different types of data or nuanced tasks".

The authors of [HW24] focus on data profiling (discovering data quality issues) with LLMs and highlight a necessity to consider not only statistical properties but also semantics. They suggest generating a Semantic Context as the first step of their profiling pipeline which is represented as "natural language descriptions of tables and columns". It is then used for making a decision on semantic acceptability of statistical discrepancies of data with an LLM.

[SBME24] framework leverages metadata for Schema Matching - "identification of semantic correspondences between elements of two or more database schemas". Each schema together with its associated metadata is used to create a document representation which is then used for selecting the top N candidates via embedding search and final attributes matching done by LLM.

3.2 Semantic Metadata for Text-to-SQL

Recent State-of-the-art Text-to-SQL techniques advance in a few different directions to improve the SQL generation capabilities. Some of them ensemble multiple generation models to obtain few query candidates and then select the best one with, for example, majority voting or another potentially fine-tuned model [PLS+24], [GGL+25], [GLL+24], [TPC+24], [LPKP24]. Another direction is to decompose the complex process of SQL generation into smaller subtasks to simplify the objective for an LLM. For instance, [PR23], [PR24] and [TPC+24] separate retrieving only the relevant parts of the schema as the first step, which is then used in the following stages. [WRY+25] leverages a similar idea but on a prompt level by firstly asking the LLM to produce "a series of intermediate steps" to provoke the Chain-of-Thought reasoning. Many approaches explore the potential of supervised finetuning [LZL+24], [PR24]. [GWL+23] aiming to close the gap between leading proprietary and smaller open source models. Some of the systems apply reinforcement learning [MZX+25], [PTS+25]. The integral parts of the vast majority of methods are the self-consistency [WWS+23] and refinement of a generated query using the execution error as a feedback. This module proves to be highly beneficial for text-to-SQL [TPC+24], [PR23], [WRY+25], [PLS+24], [GLL+24], [XXZG25], [DRX+25].

Most of the aforementioned systems imply multiple LLM calls, which typically results in increased latency and potentially cost. While [TPC+24] attempts to mitigate this issue

by minimizing API calls through its design, it still requires 6 LLM calls even in its most conservative configuration. A more lightweight approach to improve the Text-to-SQL quality involves prompt engineering. It has a limited degree of potential improvement but maintains the latency and cost reasonable for industrial deployments. Moreover, we believe that the most advanced Language Models now are already capable of solving the majority of real-life Text-to-SQL problems if given all the necessary context.

There are a few aspects that one could experiment with when prompting LLMs for Text-to-SQL tasks. [CFL23], [GGL+25] and [GWL+23] explore how to efficiently represent a schema and database content in a prompt. [GGL+25] and [XXZG25] consider the best methods to select few-shot examples of the database content. Similarly, [LWZ+24], [GWL+23], [LPKP24] and [GTT+23] study selecting the best demonstration examples for in-context learning using already existing or generated NL-SQL pairs. Finally, enriching the LLM context with semantic metadata of the underlying data and schema can also facilitate Text-to-SQL. There are, however, different metadata properties which could be added as a part of the context:

- Table and column descriptions [TPC+24], [GLL+24], [WRY+25], [LZL+24], [GGL+25], [LPKP24], [MAJM24], [SAM+24], [HDW23] - short textual explanations for table and column names. The most common type of metadata. It is included into the "Oracle Knowledge" of Text-to-SQL benchmark BIRD [LHQ+24], and almost all the submitted models make use of it[1]. It is usually added to the SQL generation prompt as a context but can also be used in other parts of Text-to-SQL pipeline: [GGL+25], for instance, incorporates it into schema linking and candidate selection modules;

- Value descriptions [TPC+24], [SZZ+23] - definition of which values the column can contain and what these actually mean. It is also included in the "Oracle Knowledge" of BIRD;

- Evidence [TPC+24], [WRY+25], [GGL+25], [LPKP24], [SAM+24] [SZZ+23], [PLS+24], [DGL+23] - external supporting knowledge about the domain. [SZZ+23] experiments with including relevant web pages and term explanations for ambiguous column names and values. [DGL+23] incorporates domain and mathematical knowledge which could be required for generating the correct SQL query: for example, when the user asks about the density, it is necessary to know that density $= \frac{\text{total\_number}}{\text{space}}$;

- Documentation of properties [HDW23], [SZZ+23], [GLL+24] - summary about particular data qualities. For instance, in [HDW23] the authors manually document three common data ambiguity issues regarding Value Consistency, Data Coverage and Data Granularity, and then demonstrate how such documentation improves SQL

[1] https://bird-bench.github.io/

generation. Another example is Measure/Dimension notion in [SZZ+23] following the usage of those in Tableau and Excel. Similarly, [GLL+24] categorizes fields as Measure, Code, Enum, Text, or Datetime, and uses these categories to generate descriptions for prompting;

- Statistical Features [SZZ+23], [GLL+24] - numerical descriptors summarizing quantitative aspects of the table. [SZZ+23] employs features from [HZZ+23] including progression type, string features, number range features and distribution features. [GLL+24] utilizes simpler features such as min/max/mean for numbers and min/max string length for strings to generate the descriptions.

Method	**Identifiers descriptions**	**Value descriptions**	**Evidence**	**Properties**	**Statistics**
CHESS [TPC+24]	✓	✓	✓	✗	✗
XiYan-SQL [GLL+24], [GL25]	✓	✗	✗	✓	✓
MAC-SQL [WRY+25]	✓	✗	✓	✗	✗
CodeS [LZL+24]	✓	✗	✗	✗	✗
MSc-SQL [GGL+25]	✓	✗	✓	✗	✗
MCS-SQL [LPKP24]	✓	✗	✓	✗	✗
Death of Schema Linking [MAJM24]	✓	✗	✗	✗	✗
SQL-PaLM [SAM+24]	✓	✗	✓	✗	✗
Tap4LLM [SZZ+23]	✓	✓	✓	✓	✓
CHASE-SQL [PLS+24]	✗	✗	✓	✗	✗
ReGrouP [DGL+23]	✗	✗	✓	✗	✗
How Documentation Improves GPT's Text-to-SQL [HDW23]	✓	✗	✗	✓	✗

Table 3.1: Semantic metadata usage in Text-to-SQL methods

All of these features can add semantic value to the prompt, however, incorporating them does not necessarily lead to an improvement of the performance. For example, the influence of incorporating table and column descriptions into the prompt has been met with conflicting conclusions across different sources. [LZL+24] and [MAJM24] report their positive impact when dealing with obscure column names of BIRD. In contrast, the authors of [HDW23] emphasize the predominant importance of other documentation types compared to column descriptions, highlighting LLMs' inherent ability to infer semantic meaning from vague names. However, it is important to note that their evaluation was based on the KaggleDBQA dataset [LPR21], which features more interpretable column identifiers than those found in BIRD. It highlights the importance of the dataset used for

evaluation.

3.3 Generating Semantic Metadata

The majority of the methods referenced in the sections 3.1.2 and 3.2 assume the existence of a data catalog containing data documentation. This is not often the case in industrial settings. Creating and maintaining high-quality metadata requires significant human effort and attention and is often disregarded in organizations. Therefore there has been a need to explore the methods of how metadata can be automatically generated.

The problem of the increasing number of digital information repositories with poor metadata was recognized early on and was initially addressed in the context of web pages, whose numbers surged dramatically with the advent of the World Wide Web. It fostered an emergence of metadata standards [WKLW98], [dHSW02] to ensure consistent formatting that would enable search among web resources. However, a very few of the proposed standards were adopted by actual users [War03]. This naturally led to research into automated methods for metadata generation. Early approaches applied text mining techniques to create semantic descriptions of Web Pages [YL05], [DCW03]. Others explore ideas of spreading metadata through collaborative filtering [RBS09]. Finally, some methods are based on ontologies [CHS04], [EMSS00].

While early efforts focused on standardizing metadata for web resources, the increasing volume and complexity of data stored in databases and data warehouses present a new dimension to this problem. It results in the introduction of standardized specification Data Catalog Vocabulary (DCAT) [W3C20], widely adopted in European government and public sector organizations [Eur19]. Many of the enterprise storage companies adopt their own metadata standards. These usually include detailed technical specifications related to data location, access patterns, redundancy levels, and data integrity. It primarily serves technical and operational purposes, and does not typically cover the semantic aspects crucial for data understanding and data discovery both by human and AI agents.

The generation of the semantic metadata became more feasible in the recent years due the development of LLMs with their advanced natural language understanding (NLU) and natural language generation (NLG) abilities. Usually it is presented as a textual summary about data and schema/part of the schema in question. The LLM-based description generation has been adopted by a few commercial companies such as Snowflake [Sno25a], Databricks [Dat25a], Dataedo [Dat25b], Collibra [Col25]. All of them utilize internal technical metadata including names of catalogs, tables, columns, location within the organizational structure, data types, notes on primary and foreign key, potentially data samples and user's input as a context. Based on the company blogs [Dat25b], [Col24], Dataedo and Collibra prompt GPT-family models through OpenAI API. Snowflake and Databricks

do not explicitly disclose the model they use; however, Snowflake charges for this feature based on credits for smaller LLMs, such as Mistral-7b and Llama 3.1-8b [Sno25b], while Databricks mentions a fine-tuned Mistral-7b in their blog [Dat24a].

A similar LLM-based strategy is applied in [HW24] and [GL25] to generate descriptions for downstream applications, namely Data Profiling and Text-to-SQL. [WHL+24] also opt to simply prompt LLM providing it with schema, column name and a few data samples. Focusing particularly on metadata for data catalogs, the authors of [SKDK25] once again employ an LLM to create table and column captions. In addition, they introduce an expander module to resolve abbreviations using a hand-crafted lookup dictionary and enhance their prompt with already verified descriptions for similar columns. Most of the frameworks also use the generated column descriptions to create table descriptions [SKDK25], [GL25], [Ano24].

One of the challenges of descriptions generation, as with any generation task, is evaluation. Different papers have explored various approaches to address it. [WHL+24] uses manual annotation categorizing each description as "Perfect," "Somewhat Correct," "Incorrect," "No Description," or "I can't tell". The quality of such annotations is high (the reported annotator's agreement was 0.61-0.68 Cohen's Kappa score) but the process is very time-consuming. [Ano24] proposes a framework where descriptions are evaluated by judging LLMs and the SelfCheck-NLI [MLG23] module which ensures the consistency of the summaries and helps to avoid hallucinations. The authors also consider "supervised" metrics, such as Coherence, Perplexity, Semantic Entropy, etc, and report high consistency of over half of the metrics with LLM-judges preferences. Nonetheless, despite a reasonable alignment with human scores, such a method is still a proxy solution to human evaluation.

Instead of evaluating the quality of the generated summary itself, an alternative approach for evaluation can focus on the performance of downstream applications. In this paradigm, one description is considered better than another if using it yields superior results on a downstream task. For instance, [GL25] and [WHL+24] evaluate the metadata based on the text-to-SQL results while [SZZ+23] also considers other tableQA tasks. It is worth noting that it might not always correspond to a human perspective. For instance, [Ano24] discovered that more concise summaries received a lower score from LLM judges while in [WHL+24] excessive descriptions with some redundant information were more beneficial for text-to-SQL then the ones human annotators deemed as 'Perfect'.

3.4 Modern Data Catalogs

Metadata is the backbone for modern data management systems, and catalogs of data are the most comprehensive way of maintaining data assets organized, discovered, and governed at scale. With more diverse data sources being gathered by companies, it has

become imperative to move away from legacy data inventories to intelligent, metadata-driven catalogs in order to facilitate data discovery and analytics.

A simple example of such transformation is described in detail in Notion's blog [Not24], which capture their evolution from unstructured and chaotic system to maintainable catalog platform synced with data science and BI tools. One of the key aspects that received a lot of attention is automatically generating, verifying and maintaining the existing tables and columns descriptions. This automation reduced the manual burden on data owners while ensuring metadata quality and consistency across the growing data ecosystem.

A more intricate system for data discovery is introduced in [AYZ+25] which demonstrates how LLMs can transform raw metadata into hierarchical, semantically meaningful catalog structures. LEDD integrates six different metadata facets, including value characteristics, sibling columns and descriptions of the columns, and uses those to form an embedding representation for each column. These embeddings are then used for the iterative clustering in combination with AI-powered summarization to create a hierarchical structures and enable semantic search across tables and columns.

The DataHub solution [Dat24b] demonstrates how AI and column descriptions enable more nuanced data discovery through knowledge graph structure, where each column node embedding is derived from its metadata. Their platform demonstrates how LLM-based entity mapping techniques can join unrelated datasets at the metadata level. By interpreting column descriptions and example values through large language models, the system learns semantic equivalences that would not be caught by traditional string-matching approaches.

Finally, a more business-oriented direction, which currently gains popularity in industry, involves constructing so called semantic layers - abstraction layers that translate technical database constructs into business-friendly terminology and metrics. Instead of schema elements, it operates business-oriented abstractions (e.g. 'revenue', 'number of active clients') unifying definitions across organization and making data more accessible to non-technical users.

Chapter 4

Methodology

In this study we aim to extract maximum value from contextual information about the schema and test its impact on SQL generation. To achieve this, we mainly focus our analysis on historical queries, from which we attempt to gain insights that could be potentially beneficial for Text-to-SQL. These insights can include deciphered identifier names, explanations of the format or the structure of the values in a specific column, notes on common usage patterns, etc. In addition to query history, we also evaluate descriptions generated from raw schema and column profiling, primarily to assess the usefulness of the information extracted from query logs using our methodology. Below, we describe how the raw information (including query history, schema and column statistics) was processed and how the descriptions were generated.

4.1 Query logs

To analyze how the schema is usually queried, we accessed a collection of historical SQL queries targeting the same databases which are used for Text-to-SQL translations. It is important to note that the queries used as a history are not a part of the test dataset for SQL generation. We explore two approaches for incorporating query history information into the column descriptions. The first one relies heavily on an LLM prompted to describe how specific column is used in a sampled query, which are then aggregated and used in a follow-up LLM prompt to generate a final description.The second approach involves manually analyzing the historical query data by parsing the queries into smaller structural elements and then revealing the most frequent patterns by aggregation. The following subsections describe both of these methods in detail.

4.1.1 Query Annotation

The first approach we test uses an LLM to analyze queries that reference the column in question. First, for each column, we randomly select N SQL query samples from the history. Then, we prompt an LLM to describe how the column is used and what its purpose is in each query. The results are then aggregated in a list which is then used to generate a description for the column. A high-level overview of this method is displayed by Algorithm 1.

Algorithm 1 Generating a Column Description using Query Annotation

```
for every column do
    Retrieve a list of N queries that reference the column
    for each query in the retrieved list do
        Prompt an LLM to annotate the column's usage in the query
        Add the LLM's annotation to the list
    end for
    Use the list of annotations to prompt an LLM to generate a final description
end for
```

Here is an example of resulting list of annotations:

```
Query annotations for column 'year' in table 'races'


  1. It filters the results to include only records from a specific year.

  2. It filters race records, restricting the results to those that occurred in
     a particular year.

  3. It is retrieved as part of the final result set, providing the year of the
     race for the selected record.

  4. It filters race records to a specific year, narrowing down the results to
     races held in 2011.

  5. It groups records to count associated events per distinct period.  The
     query then identifies and returns the period with the highest number of
     events.
```

Figure 4.1: Examples of column `races.year` usage in queries

While this method is straightforward and requires little engineering effort, it incurs additional cost due to LLM usage and its effectiveness hinges on the representativeness of the sampled queries. An insufficient sample set may provide false signals on what actually is a common pattern and lead to incorrect judgements. One way to prevent this is to increase the number of samples but it directly increases the cost, an alternative would be to experiment with selecting the right example queries. In this study we will utilize 5 randomly selected samples to assess feasibility of such an approach, reserving a more sophisticated sampling strategy for future work.

4.1.2 Query Pattern Mining

Analyzing historical queries can reveal a wealth of information about the underlying schema. The most immediate insights include **primary-foreign key relationships and joinable tables**. As just a part of metadata, this information can often be lost or poorly maintained in the industrial setting. However, in many scenarios knowing how to join the tables is essential for correctly translating the user's request into SQL.

Secondly, query logs capture established **usage patterns**. For instance, one can learn if a certain column is mostly used for filtering while another is often selected or used as a tag to group records. Such insights can help identify a column's type in the measure/dimension dichotomy, a concept used in several modern data management frameworks. Going deeper into the analysis, we can also extract **prominent expressions** with specific columns. This not only provides simple hints about how a column should be processed, but also reveals more complex expressions that carry significant business value, proving extremely beneficial when building a semantic layer on top of the data.

Finally, we can also infer patterns in the **co-occurrence of columns and tables**. While perhaps less helpful than the previous two from a data management perspective, this can positively impact Text-to-SQL translation, especially when dealing with ambiguous user questions. For instance, if a user asks for a "date", it might be unclear if they expect only the calendar date or also the time. Knowing how similar requests have been handled historically can lead to more accurate answers, which is especially useful for benchmarking. Additionally, it can come in handy for schema linking: in this scenario when recall is crucial, knowing that two columns are always used together provides a strong signal to include both, even if only one of them is discovered by the schema retrieval method.

While additional information such as data lineage could also be extracted, we focus specifically on these three aspects when analyzing query history. To enable this analysis, we parse the queries and decompose them into smaller structural elements. For each query, we identify the referenced columns and tables, along with the clause types in which they appear. In addition, we also identify all the expressions occurring in the query. We then aggregate this information to identify the most frequently occurring patterns. The subsection below focuses on the technical details of implementing parser and the aggregation technique.

First-stage parsing

The first step of our pipeline is to parse the queries into smaller, operable components. There have been a few options that we considered for this step. The first one was to process physical plans of queries produced by DuckDB's `EXPLAIN` statement. A big advantage of this approach is that it binds the query to the database and recovers all the column

projections automatically which can be challenging to recover manually when it comes to complex convoluted queries full of Common Table Expressions or subqueries with aliases. It also avoids the need to manually retrieve and process the schema to expand `SELECT *`. Nonetheless, the plan from `EXPLAIN` does not capture all expression details, e.g., `CASE` expressions are referenced like simple columns. This was a sufficient reason for us to abandon this method.

Another approach was to use existing SQL parsing frameworks. The most viable option supporting DuckDB dialect would be the `sqlglot` library [1]. However, its parsing quality for DuckDB queries is still imperfect. Furthermore, since it is fully written in Python and operates on a large internal object structure, using sqlglot would require implementing our parsing module in Python as well which we aimed to avoid.

Finally, we explored the idea of traversing the Abstract Syntax Trees (AST) of queries produced by DuckDB native function `json_serialize_sql`. Despite not doing the binding step and thus not disambiguating all the columns, it offers a comprehensive structure of raw blocks in an easily readable format (JSON) which we could build on top of.

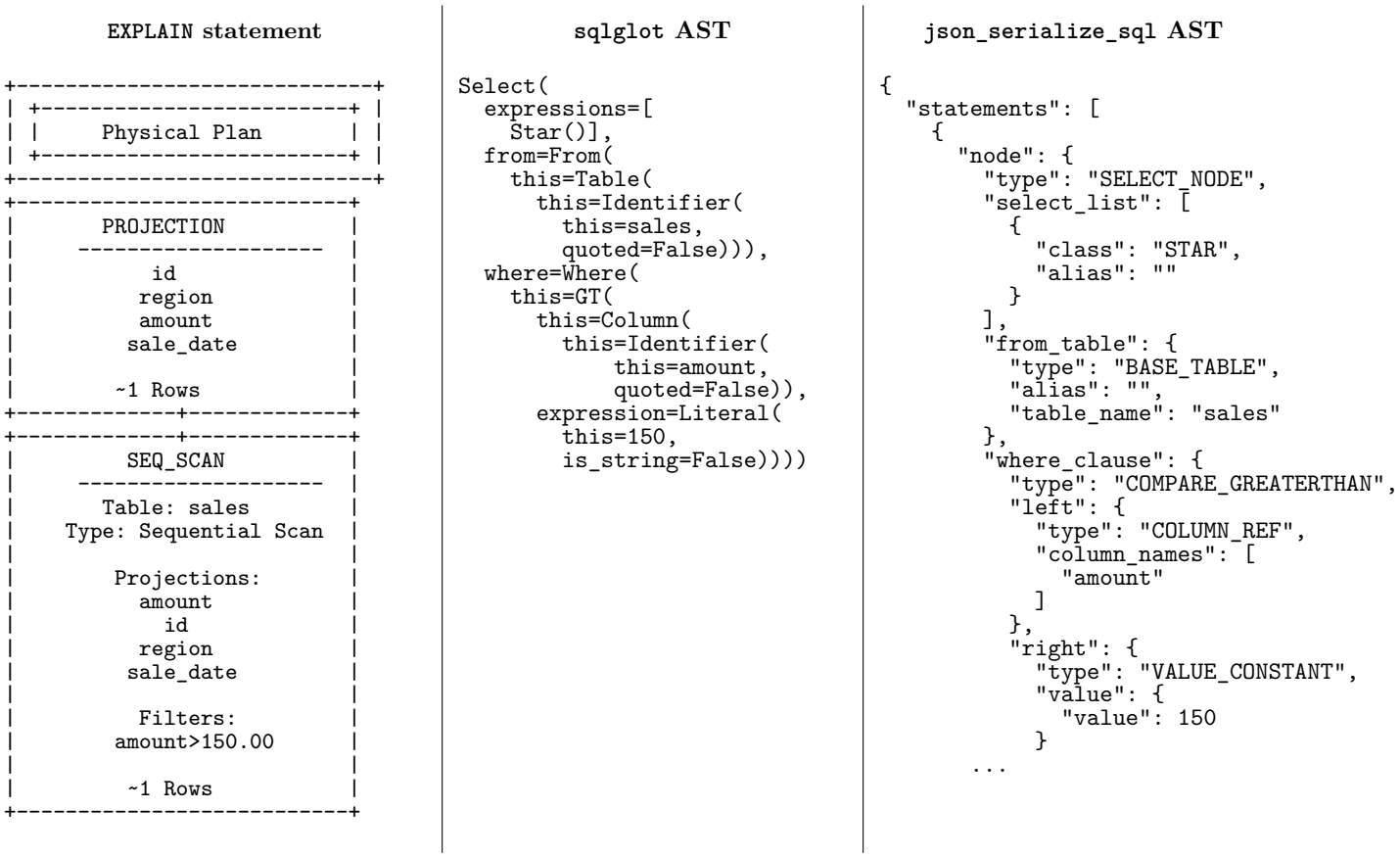

Figure 4.2: Representations of the query `SELECT * FROM sales WHERE amount > 150;`

[1] `https://github.com/tobymao/sqlglot`

Ultimately, we decided to combine the latter two methods. We used `sqlglot` to qualify the queries which, given a schema, resolves the identifiers in a query to their full name and replaces the `*` in `SELECT *` with the actual columns. We then applied `json_serialize_sql` to the qualified queries and used the resulting ASTs as a foundation for extracting column/table occurrences and expressions.

Parsing query ASTs - considerations

Traversing an AST in JSON format is relatively straightforward. However, since our goal is to track every mention of a specific column or table in the query, the main challenge lies in resolving aliases and identifying the actual columns in more intricate queries. For instance, in the query 4.3 the `SELECT` of the main clause includes **`product_101_revenue`** which is not a real column but an alias for the expression `SUM(sale_amount) FILTER (WHERE product_id = 101)`, and thus needs to be resolved. In our parser, each mention of **`product_101_revenue`** should add an occurrence for **`sales.sale_amount`**, **`sales.product_id`** as these are part of the expression.

```
WITH monthly_revenue_metrics AS (
  SELECT
    SUM(sale_amount) FILTER (WHERE product_id = 101) AS product_101_revenue,
    SUM(sale_amount) AS total_revenue
  FROM sales
  WHERE
    sale_date >= CURRENT_DATE - INTERVAL '30 days'
)
SELECT
  product_101_revenue,
  total_revenue,
  (product_101_revenue / total_revenue) * 100 AS
     percentage_contribution
FROM monthly_revenue_metrics;
```

Figure 4.3: Example SQL Query with challenging column processing

Moreover, if **`product_101_revenue`** is used in another expression, we also aim to replace it with its resolved form inside the expression. This is necessary for more accurate aggregation later on: the same expression can be aliased differently, so resolving it to its canonical form ensures that semantically equivalent expressions have the same representation after parsing, even if written differently. For the same purpose, instead of using the same form as in the query, we rewrite all the expressions using standardized syntax. DuckDB syntax allows many semantically equivalent variants of the same functions. For example, `LIKE` and `~~` are equivalent, as are `json_extract(j, '$.family')` and `j->'$.family'`, or simply `a + b` and `b + a`. We aim to normalize these to the same

representation after parsing.

Recovering actual columns gets more difficult with nested CTEs or subqueries. Each clause can project columns from tables, subqueries or CTEs, and the scope of available sources varies at each nesting level. There can be conflicting names between the sources from levels as shown in the example-query 4.4: in the main `FROM` clause you can't refer to the CTE **`inner`** but there might well be a different table with the same name **`inner`**). For th is important to track the scope of available sources at each level to recover columns accurately.

```
WITH outer AS (
    WITH inner AS (
        SELECT ... FROM table
    )
    SELECT ... FROM inner (CTE)]
)
SELECT * from inner (actual table, can't be the CTE)
```

Figure 4.4: Example SQL Query with conflicting sources names

Moreover, it is also important to account for CTEs defined at higher levels. For instance, in the example X the **`inner`** CTE can reference the **`outer_1`** CTE, even though it is one level above. This only works in one direction: it is not possible to reference the **`inner`** CTE from outside.

```
WITH
    outer_1 AS (...),
    outer_2 AS (
        WITH inner AS (
            SELECT ... FROM [<any_table> | outer_1]
            )
        SELECT ... FROM [<any_table> | inner | outer_1]
    )
SELECT * FROM [<any_table> | outer_1 | outer_2]
```

Figure 4.5: Example showing CTE visibility across nesting levels

Parsing query ASTs - Implementation

In this work we design a custom AST traversal algorithm which takes into account these challenges. It currently parses only `SELECT` statements, a limitation inherited from `json_serialize_sql`. However, this is not critical for our experiments, as the target information we look for is primarily found in `SELECTs`. The schematic overview for our algorithm is shown in the Figure 4.6.

Figure 4.6: AST traversal overview

For every `SELECT` statement, the algorithm traverses the tree starting from the data sources: CTEs and `FROM` clauses. It defines all available sources (CTEs + tables from FROM clause) at the current level and also considers the CTEs from outside. Every source at each level is tracked through a nested lineage structure constructed in parallel with the tree traversal and level-wise resembling the AST. CTEs are parsed recursively first: the same `parse_select` function is called for every `SELECT` statement inside CTEs which in turn prioritize its own CTEs. The `FROM` clause is parsed only after all the CTEs at the current level have been processed. This ensures a bottom-up traversal making it possible to conclude that any table mentioned in the `FROM` clause that is not in the lineage (and thus not a CTE) is an actual table, and we can add an occurrence record for it. One limitation of this approach is its reliance on users writing CTEs sequentially: a first CTE can be referenced in a second or third, but usually not vice versa. If a CTE is referenced before it's defined as demonstrated in Example 4.7, the algorithm would incorrectly identify it as a physical table. However, this is an unnatural way of writing queries that we've never encountered in our training data, so we don't consider it a critical issue. Alongside table sources, we also parse the joined tables and subqueries in a `FROM` clause.

```
WITH cte_1 AS (
    SELECT item FROM cte_2
),
cte_2 AS (
    SELECT item FROM table
)
SELECT item FROM cte_1;
```

Figure 4.7: Example showing CTE forward reference

As mentioned above, in parallel with the tree traversal the algorithm also builds a similarly structured data lineage (these are also illustrated in Figure 4.6. It keeps track of the CTEs, tables, columns and expressions relevant to the current level. During processing the sources, we only update the CTEs and table fields: the tables are just inserted into the lineage as end nodes while the CTEs are stored as recursive dictionaries where the key is the CTE name and the value is the lineage at a deeper level with its own CTEs, tables, columns and expressions.

After processing all sources, every column at the current level must have its source in the lineage. In case a referenced column is an alias of another from a CTE or a subquery, the initial form of the column (its actual name and table it refers to) can be restored by recursively going down the lineage as all the CTEs have already been processed. This process is shown in Figure 4.8. It allows us to begin parsing all the other clauses that

reference columns: `SELECT`, `WHERE`, `GROUP BY`, `HAVING`, and the `ORDER BY` and `DISTINCT` modifiers.

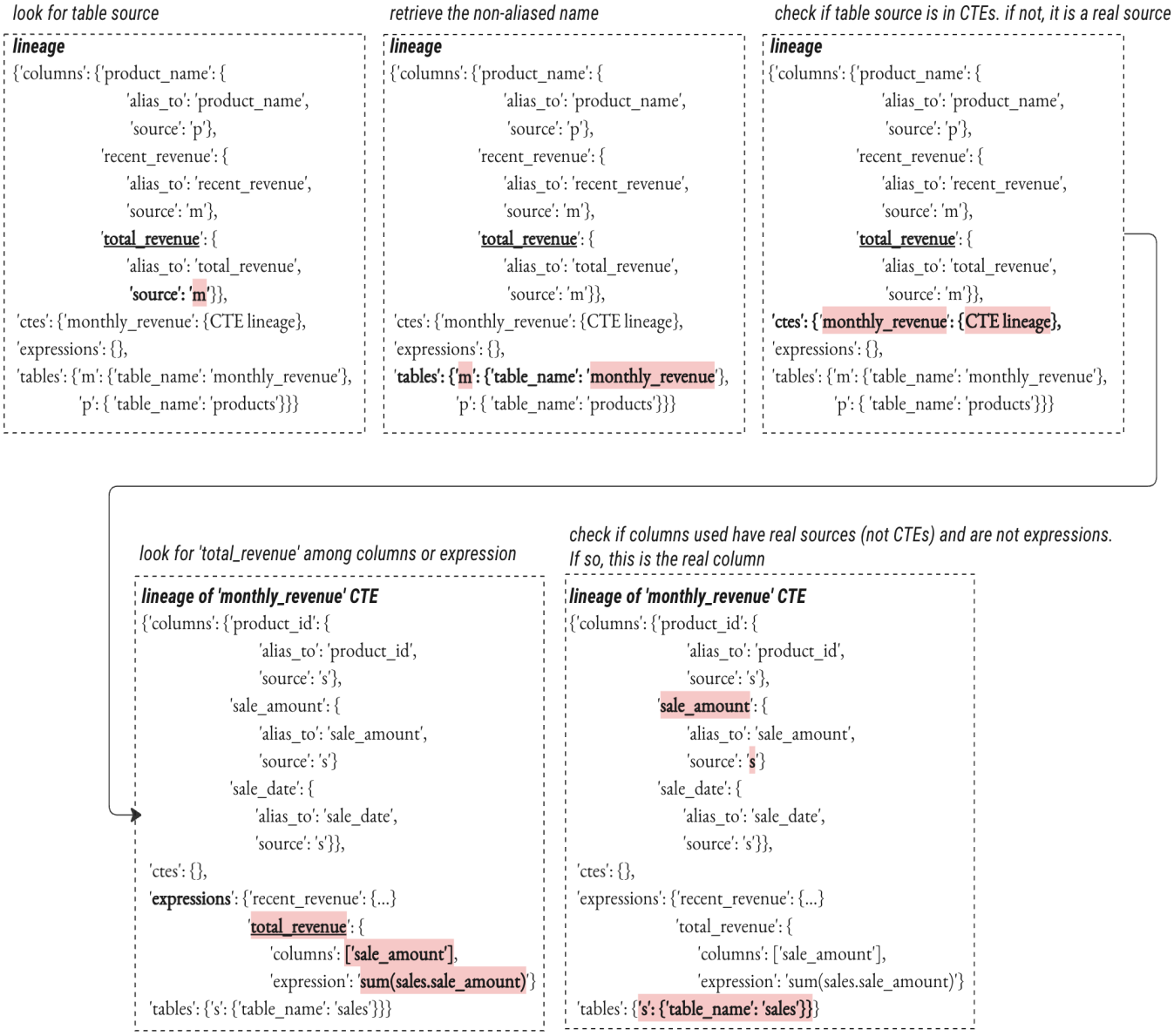

Figure 4.8: Recovering the initial column form through the lineage for `total_revenue` column

Within a clause, the column may be referenced as a part of an expression. Multiple layers of operators or functions may be applied to the column value. That is why every expression reference node is parsed recursively until a column reference or a constant is found. When an end node is a real column, we add an occurrence record for it noting the

type of a clause it was discovered at; otherwise, the actual columns are restored through the lineage and the occurrence is added for each of them. The lineage is updated at this stage with the used columns and expressions. Finally, as the parsing moves back up the tree from the column reference to the initial expression, a normalized representation of that expression is constructed on each layer and added to our output (see Example 4.9).

Original Query:
```
SELECT sum(server.cost)::INT AS total_blended_cost
FROM server GROUP BY ALL
HAVING total_blended_cost > 1000
```

Expression with layers:
```
CAST(sum(server.cost) AS INTEGER) > '1000'
```

Normalized derivations:
```
sum(server.cost)
CAST(sum(server.cost) AS INTEGER)
CAST(sum(server.cost) AS INTEGER) > '1000'
```

Figure 4.9: Normalized expressions extracted from the example query

The algorithm outputs a set of all real identifiers and every single expression mentioned in a query. The parsing logic is entirely written in Python, and is then encapsulated within a DBT Python model for streamlined processing. After parsing, both occurrences and expressions are stored in two separate relational tables ready for further analysis.

Aggregation

After decomposing the query into its constituent column and table references, as well as expressions, we can identify common trends by aggregating the results. As mentioned previously, for each column or table we aim to investigate its join relationships, the types of clauses it typically appears, co-occurring identifiers and relevant expressions. To achieve this, we collect the following information for each identifier:

- **General frequency** – how many unique queries use this identifier;
- **For tables:**
  - **Joined tables** – which tables can be joined with the current one, and how frequently. Queries don't have to be unique;
- **For columns:**
  - **Frequency in each clause** – which types of clauses the column appears in and how often. The queries don't have to be unique;

- **Co-occurrence by clause type** – which columns appear alongside the current one within the same unique clause, and how often;
- **Expression in each clause** – which expressions use the current column and how frequently they occur;

To enable the model to reason about the frequency of certain events relative to the entire dataset, we use a z-score or a standard score as a frequency metric. All extracted information is structured as text which is then used to generate the description. Figure 4.10 displays an example of how it looks like in the prompt:

Statistical analysis for column 'Consumption'

Column `Consumption` is used in 14 SQL queries, which represents a z-score of 0.54 compared to the usage frequency of other columns.

Below is the info about the clause types where the column `Consumption` usually appears, the corresponding number of occurrences and the z-score (calculated over how frequently this specific column appears in different clause types compared to other columns):

Clause Type	Query Count	Z-score
ORDER_BY	7	1.02
SELECT	6	0.70
WHERE	1	-0.86
HAVING	1	-0.86

When this column appears in `ORDER BY` clauses, it is typically used alone with no other columns around it.

Here is the info about the columns that often occur together with `Consumption` in `SELECT` clauses, the corresponding number of queries where they were found together and the z-score (indicating how often it appears alongside the target column in comparison to other co-occurring columns):

Column Name	Query Count	Z-score
yearmonth.customerid	2	1.15
transactions_1k.customerid	1	-0.58
yearmonth.date	1	-0.58

These are some common `SELECT` expressions which use this column and which were commonly present in SQL queries in the past:

Expression	Query Count	Aliases
sum(yearmonth.consumption)	3	
sum(CASE WHEN yearmonth.customerid = '7' THEN yearmonth.consumption ELSE '0' END)	1	
CASE WHEN yearmonth.customerid = '7' THEN yearmonth.consumption ELSE '0' END	1	
(nullif('12', '0') / sum(yearmonth.consumption))	1	monthlyconsumption

Here is the info about the columns that often occur together with `Consumption` in `WHERE` clauses, the corresponding number of queries where they were found together and the z-score (indicating how often it appears alongside the target column in comparison to other co-occurring columns):

Column Name	Query Count	Z-score
yearmonth.date	1	N/A

These are some common WHERE expressions which use this column and which were commonly present in SQL queries in the past:

Expression	Query Count	Aliases
yearmonth.consumption = '21458217' AND yearmonth.date = '201306'	1	
yearmonth.consumption = '21458217'	1	

Figure 4.10: Information extracted from query logs for the `yearmonth.Consumption` column

4.2 Baseline Methods: Schema & Column profiling

While our study is mostly centered around query history, we also conducted a few control experiments with descriptions derived from just schema and a combination of schema and profiling information. The schema was always presented in a prompt as a `CREATE TABLE` statement and included names of the identifiers, data types and all the constraints regarding certain identifiers that have been documented (in case with our datasets those were primary/foreign key as well as `(NOT) NULL` and `UNIQUE` constraints).

As part of the column profiling information, the following attributes are considered for each column:

- Column's data type;
- Total number of rows;
- Uniqueness — whether all values in the column are unique;
- Null values — whether the column contains nulls and, if so, the percentage of null values;

While the data type is often inferable from the schema and values, notes on the uniqueness and the presence of null values can be extremely useful to indicate a necessity of using a `NOT NULL` check or a `DISTINCT` clause.

Inspired by [SSJG25], [SZZ+23], we also consider data-type-specific statistics. For string-type columns (`VARCHAR`), additional profiling includes:

- Length statistics: minimum, maximum, and average string lengths;
- Character composition analysis: counts of values containing only digits, only uppercase letters, only lowercase letters, or mixed character types;
- Distinct value count and its percentage of total rows;

For numeric columns (`INTEGER`, `FLOAT`, `DECIMAL`, etc.), the profiling extends to simple statistical summaries including minimum, maximum, and average values, providing the insight into the data distribution and scale.

Figure 4.11 demonstrates examples of string and numeric column profiles we extracted:

4.3 Generating Descriptions

Following the established trend [WHL+24], [GL25] [Ano24], [SKDK25], we also use LLMs to produce a natural language description for each table and column. Due to the diverse

```
organizations.alternate_billing_provider
Data type:  VARCHAR
Total rows:  XXX
Is unique:  False
Contains nulls:  Yes (98.58%)
String length:  min=6, max=6, avg=6.0
Distinct values:  1 (0.0% of total rows)
```

```
slo.success_ration
Data type:  DOUBLE
Total rows:  XXX
Is unique:  False
Contains nulls:  No
Numeric range:  min=0.879988, max=1.0, avg=1.00
Distinct values:  772 (0.7% of total rows)
```

Figure 4.11: Examples of string and numeric column profiles

nature and large size of the extracted information, we expect the LLM to summarize the input, highlighting its most valuable aspects. In addition, we rely on the intrinsic knowledge of language models, for example, to infer the format of values from a few samples or deduce unclear table and column names (which LLMs have been reported to perform quite well [ZSS+23]). Resolving these ambiguities as a separate step simplifies the main task and can be especially beneficial when a less advanced LLM is used for SQL generation.

Our goal in this study is to assess the usefulness of the generated metadata for Text-to-SQL and demonstrate the impact it can make. That is why we opt to use a strong LLM for generating the descriptions. After preliminary experimentation with GPT-4 and Gemini 2.5 families, we decided to use **Gemini-2.5-flash-preview-05-20**, one of the more advanced proprietary models from Google DeepMind, striking balance between quality and latency.

In our experiments, we incorporated various metadata sources into the prompt to obtain different types of summaries. The `CREATE TABLE` statement of the current table was always included as a context. The table below outlines our experimental settings along with the examples of the generated descriptions:

It is worth noting that the generation prompt varied slightly across experiments. The main differences laid in the instructions on how to interpret the provided information and what details the model should focus on. For example, there's no point asking the model to pay more attention to joinable columns when the prompt only includes the column profile. The core message, however, remained consistent: to generate a precise description that includes the primary purpose of the identifier and any potentially useful additional information. All of the prompts we used can be found in the Appendix A.

Name	**Metadata used**	**Example**
Schema	Only schema provided (names + data types + documented constraints)	*Stores the measured consumption value for a customer on a specific date, primarily used to track and quantify usage over time. This column is a double-precision floating-point number and can be null if consumption data is unavailable*
Stats	Schema and column profiling	*Represents the measured quantity of consumption for a customer on a specific date. Values are numeric and can be negative.*
Query Annotation	Schema and summaries of random queries	*Stores a numeric value representing consumption, which can be NULL. This column is frequently aggregated by summing values per customer to calculate total or average consumption. These aggregated values are commonly used for ranking customers, identifying top consumers and ordering results. It is also utilized in conditional sums and difference calculations between aggregated values for specific customers or dates.*
Query Patterns	Schema and statistics from logs	*Represents the consumption value, stored as a double-precision floating-point number that can be null. This column is frequently used in 'ORDER BY' clauses, often independently, to sort results by consumption. It is also commonly selected, sometimes in conjunction with 'CustomerID'. A common aggregation is 'SUM(Consumption)', which calculates the total consumption. Less frequently, it appears in 'WHERE' clauses, typically used directly without expressions, sometimes alongside 'Date'.*

Table 4.1: Examples of generated descriptions for the `yearmonth.Consumption` column

Chapter 5

Experimental setup

This section outlines our evaluation setup for Text-to-SQL, including the datasets used, model configuration, and evaluation criteria.

5.1 Data

We conducted our experiments on two datasets: BIRD, a widely used benchmark for Text-to-SQL evaluation, and MDW-AMBIG, a custom dataset we built from real-world user queries over actual tables, preserving some complexities of a production-level setting.

5.1.1 BIRD

The BIRD benchmark [LHQ+24] is a large-scale Text-to-SQL dataset and currently one of the main standards for evaluation. It spans a wide range of domains and is designed to be more challenging than Spider [YZY+19], with longer and more complex queries and richer schemas. Each database is stored in SQLite format and comes with additional metadata such as example values, value descriptions, and evidence fields linking questions to relevant schema elements. In this study, we exclude this metadata, focusing solely on insights retrieved from the schema, data context, and queries. This choice better reflects a real-world setup where value descriptions and evidence fields are often unavailable.

The dataset is split into train and development sets with no overlap in databases, meaning that the dev databases have no queries executed against them in the train set. Since many of our experiments rely on the existence of historical queries, this setup prevents us from using the train set to accumulate information for descriptions. That is why we use the BIRD Mini-Dev[1] subset of 500 samples as our test set, while the remaining 1,039 queries become query history. This subset, provided by the creators of BIRD, roughly

[1] https://github.com/bird-bench/mini_dev

preserves the distribution of databases but exhibits a noticeable shift towards moderate queries in terms of query complexity compared to the original dataset.

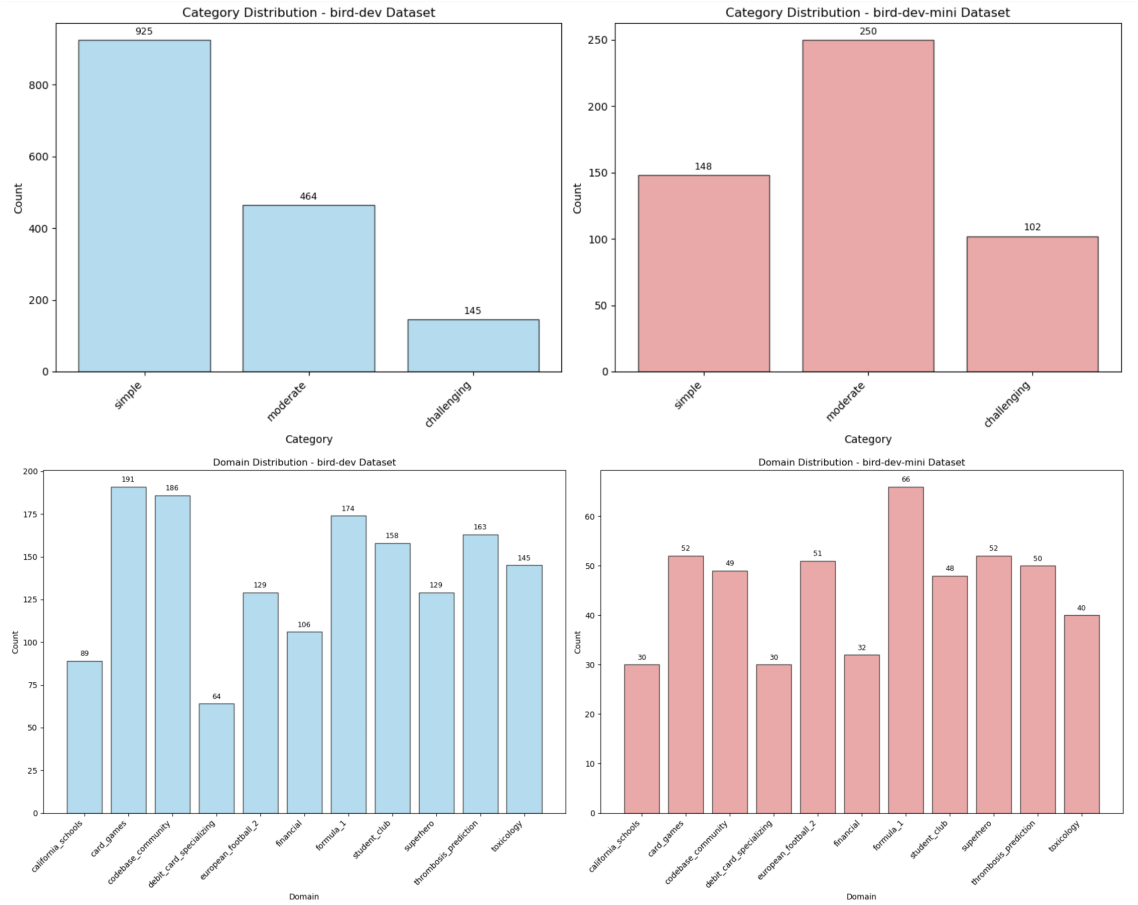

Figure 5.1: BIRD and BIRD-Dev-Mini category and domain distributions

5.1.2 MDW-AMBIG

While using BIRD as the de facto benchmark, we also created a custom dataset based on real user queries submitted through the MotherDuck UI and targeting MDW, the internal MotherDuck database used as the main storage for company information. We built this dataset for two main reasons:

- to evaluate our method in a setting close to a real-world environment, and
- to perform a more qualitative analysis of issues such as ambiguous column names and non-obvious usage patterns, which occur frequently in practice and are where we expect our method to be most effective.

To construct the dataset, we filtered all available user queries targeting only the MDW database and split them into train and test sets. This split was done by date: all queries executed before were used for training, totaling 6631 individual items, while the test set was selected from the remaining queries submitted after. We selected only queries executed through the MotherDuck UI to ensure we included only manually written queries and excluded any automatically generated ones.

Since we expect descriptions to be particularly helpful for ambiguous identifiers, we focused on queries that contained them. We used an LLM-assisted process: we provided the LLM with the schemas for all tables and prompted it to identify vague or potentially confusing column names - those that could be mixed up with others or were unclear without additional context. For each such column, we extracted all queries from the test subset that referenced it. We then filtered these queries by length (20-200 characters) to keep those that were neither trivially simple nor extremely complex; this is also roughly the size of a query that users would usually request a Text-to-SQL model to produce - more intricate questions require more thoughtful prompting which is also an effort while a simpler ones are faster to write manually. Queries with execution times longer than one minute or the ones which returned empty results were also removed. Finally, we manually reviewed the remaining queries, removing similar ones and making minor adjustments for verification and clarity. The resulting collection consisted of 50 quality queries executed against 12 different schemas.

5.1.3 BIRD and MDW-AMBIG comparison

Despite the BIRD authors describing it as a benchmark for "real-world applications" from our observations it still differs noticeably from what is seen in industrial environments. These differences appear in both the database characteristics and the query patterns.

Database Characteristics

From the data perspective, there are a few important differences between MDW and BIRD databases. The first one concerns the data type distributions demonstrated in Figure 5.2. One notable distinction is that while more than half of values in BIRD is numeric, which are straightforward to process, the main constituent of MDW are `VARCHAR` values, often requiring casting or some extra non-obvious processing. Another difference is the presence of composite types in MDW such as `STRUCT` or `ARRAY` (5.4% of all values), which are absent in the BIRD databases. These types can pose challenges for Text-to-SQL; for example, arrays often need to be unnested or iterated over when filtering by a contained value or `STRUCT` fields may need to be accessed or cast individually before use in conditions or joins.

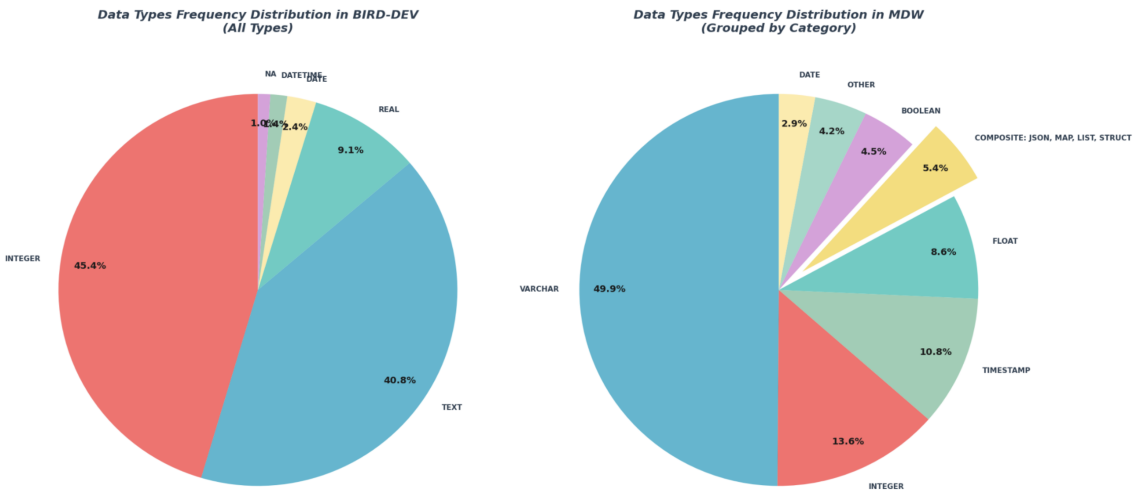

Figure 5.2: BIRD databases and MDW Data Types distribution

Constraints are also different. In BIRD, every table includes explicit metadata about primary keys and `NOT NULL` constraints. These are embedded in the DDL statements used for Text-to-SQL prompting. On the contrary, in MDW across 162 tables and 2805 columns, there are only 6 primary key constraints, 13 `NOT NULL` constraints and no foreign keys declared. This means that schema metadata is far less explicit, requiring the model to infer relationships and constraints during translating.

The vocabulary of identifiers is more domain-specific in MDW, with many company-specific terms which might not be in the training data of the model and thus be not well understood by the model. For example, *hatchling* as a standalone customer-facing application, *duckling* as a personal DuckDB instance, *jumbos*, *pulse*, or *gigas* as types of ducklings. BIRD identifiers also includes some domain-specific terminology, but its domains are generally more familiar and widely understood. Moreover, identifier names in MDW tend to be more similar on average. Figure 5.3 shows the pairwise similarities of 50 random column names within the same schema for BIRD and MDW, with MDW exhibiting higher values. Examples of particularly problematic cases are:

- columns `organization_name` vs. `organization_friendly_name`
- tables `active_organizations_daily` vs. `daily_organizations_stats`

which are frequently confused by Text-to-SQL models in practice.

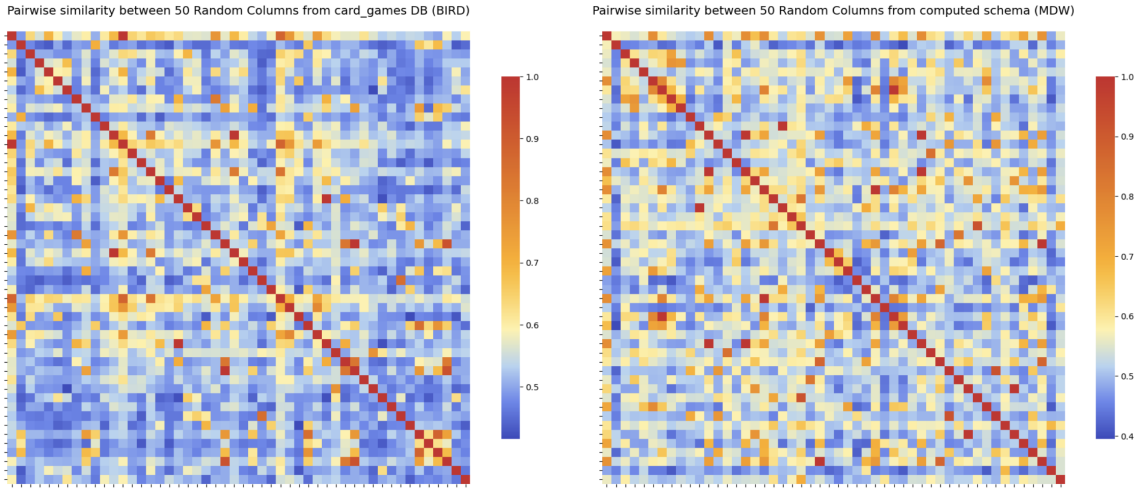

Figure 5.3: Similarity between column embeddings for BIRD and MDW. We used cosine similarity as a similarity metric, abd Google's TEXT-EMBEDDING-004 model to generate embeddings. Warmer colors indicate higher similarity values.

Queries

Importantly, the way people query MDW differs from the BIRD collection of queries. Figure 5.4 depicts the distribution of columns used across the BIRD and MDW-AMBIG datasets. As shown, in the BIRD dataset, almost 70% of columns are used across the queries, and the distribution between them is more equal, whereas MDW-AMBIG queries reference less than 60% of columns, and usage is concentrated on a small set, resulting in a steeper distribution curve. It means that when querying MDW, users tend to target the

same areas of data (narrow set of tables and columns), resulting in more repetitive query patterns compared to BIRD.

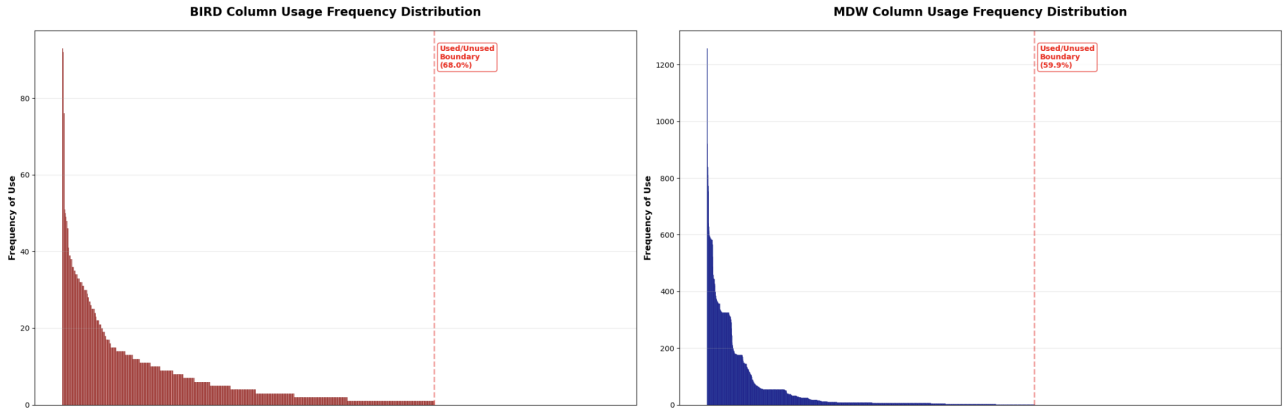

Figure 5.4: Column usage distribution for BIRD-Dev and MDW-AMBIG train sets

Moreover, not only columns are getting targeted more repetitively, but also the expressions involving those columns. In MDW-AMBIG, over 87,000 expressions extracted by our parser collapse to only 7.7k unique ones. In contrast, in the BIRD-Dev training set less than 4,000 expressions are reduced to just 2,274 unique ones. It means that on average each expression appears roughly twice in BIRD-Dev queries in comparison to 11 for MDW-AMBIG queries. In addition, the nature of expressions also differs: BIRD-Dev queries are dominated by simple comparisons while MDW-AMBIG queries use a broader variety of functions and operators as demonstrated in Figure 5.5.

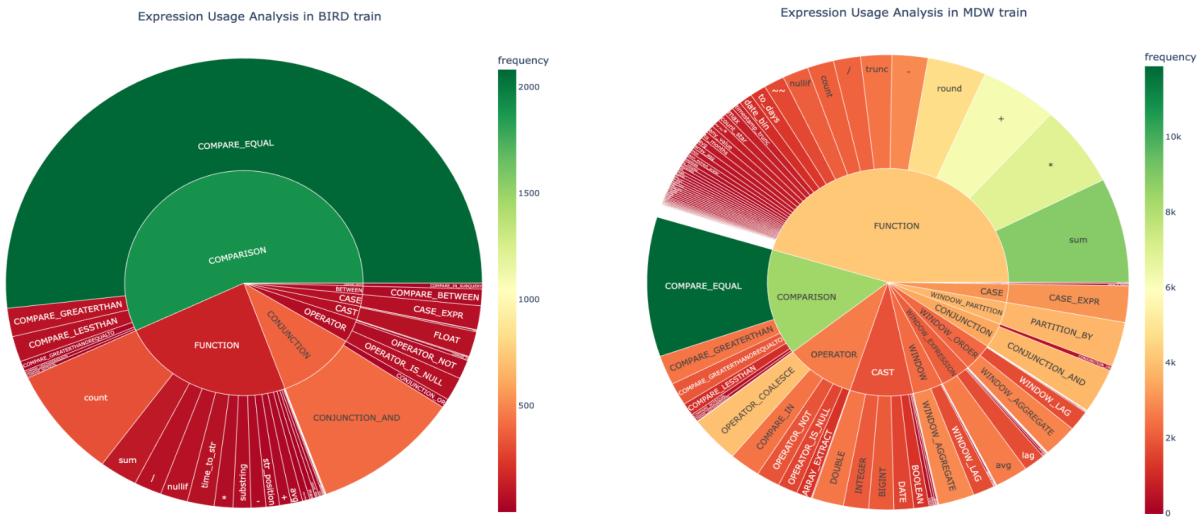

Figure 5.5: Expression types in BIRD-Dev and MDW-AMBIG train sets

In general, queries to MDW tend to be more repetitive. This may be due to users iterating on the same query to refine it or running routine daily or weekly analytics with only slight modifications, such as minor filter changes. As a result, queries in MDW-AMBIG are more clustered, whereas the BIRD-Dev dataset contains a more diverse set of queries, as illustrated in Figure 5.6.

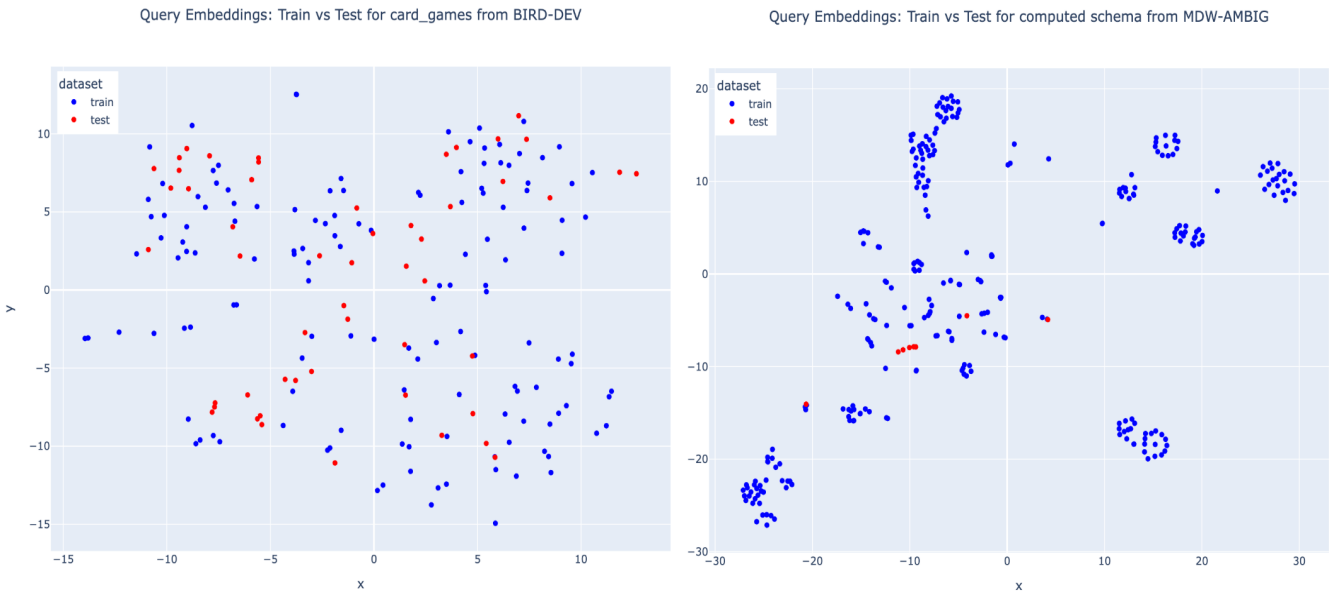

Figure 5.6: Embeddings for queries from BIRD-Dev and MDW-AMBIG. We used Google's text-embedding-004 model to generate embeddings, then applied PCA to reduce the dimensionality to 50 dimensions, followed by t-SNE for visualization in 2D.

Schema sizes also vary, with MDW having considerably larger schemas. A single schema among the ones used in MDW-AMBIG test queries on average contains 12 tables and 213 columns, whereas for BIRD, these numbers are only 7 tables and 72 columns. This has direct implications for Text-to-SQL, since the schema is always included in the prompt, increasing the context size and making the task more difficult. It can become even more challenging when the same query references different schemas, causing the number of candidate tables and columns to grow substantially.

Finally, since the number of queries for MDW is significantly higher, the train to test ratio differs between the datasets. We have 6331 individual queries as a train history for 50 test samples in MDW-AMBIG, while BIRD has only 1,000 "historical" queries for 500 test samples. This results in more consistent recurring patterns in MDW and potentially more diverse and informative schema descriptions.

5.2 Models

For Text-to-SQL translation, we experiment with two models. The first is **Gemini-2.5-flash-preview-05-20**, the same model used for generating the descriptions. As mentioned above, Gemini 2.5 is Google DeepMind's latest model lineup, offering "advanced reasoning through thinking, long context, and tool-use capabilities." With a context window of up to 1M tokens, it is reported to perform strongly on coding and reasoning benchmarks. The Gemini 2.5 family was officially released in June 2025, with preview models gradually rolled out starting in March 2025. We choose the Flash variant over Pro as it is more than twice as fast, making it more suitable for interactive in-editor assistants—features that are currently under active development at MotherDuck. The second model we evaluate is **Qwen2.5-32B**, an open-source model from Alibaba, specifically trained for code-related tasks and released in November 2024. It is the largest model in the Qwen2.5 family, with 32 billion parameters and support for up to 128k input tokens. The model is widely adopted, with over 150k downloads on Hugging Face. It ranks first on multiple datasets from the OmniSQL benchmark [LWZ+25] among non-fine-tuned open-source LLMs with more than 32B parameters.

Both models represent the current frontier in LLM capabilities, allowing us to compare a state-of-the-art commercial system with a strong open-source alternative.

The temperature for all the experiments was set to 0.

5.3 Evaluation

To measure the quality of SQL generation, we use the execution matching score (EX). This metric evaluates the correctness of the query based on its execution results rather than the exact structure or syntax of the query itself. To compute the score, both correct and predicted queries have to be executed. At this stage, we account for the discrepancy between the BIRD's ground true queries written in SQLite dialect and the DuckDB generated queries. Each query is executed against the corresponding database. There are a few examples in the dataset where this discrepancy becomes critical - specifically in queries that use `ORDER BY` in combination with `LIMIT`, and where multiple rows share the same sort key. In such cases, some of these equally ranked rows may fall within the limit, while others may be excluded, and it is inconsistent between the SQLite and DuckDB. These cases were reviewed manually.

The final score represents an average across the whole dataset where '1' is assigned if the results coincide, and '0' if they do not.

5.4 Text-to-SQL Setup

In our experiments, we adopt the simplest Text-to-SQL architecture with LLMs. We decided to not include neither schema filtering nor self-correction modules and primarily focus on the benefits from context enhancement. The overview of the architecture we used is shown in Figure 5.7.

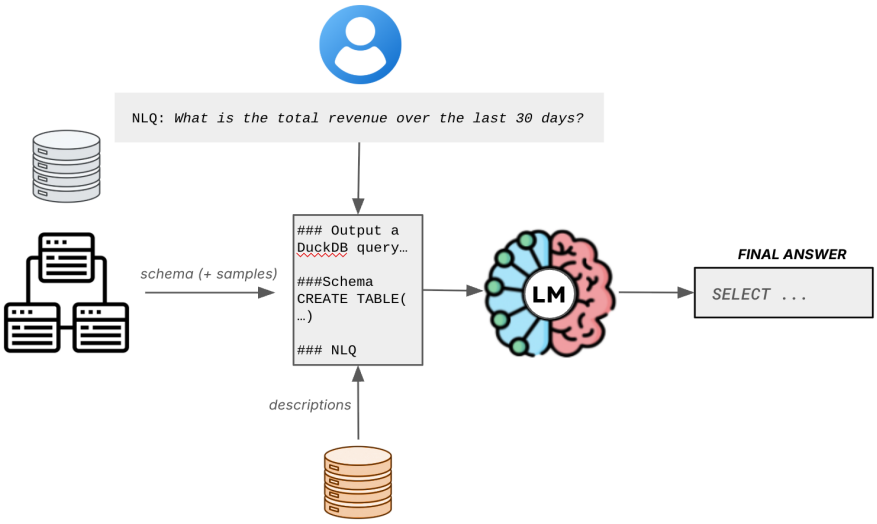

Figure 5.7: Text-to-SQL architecture in our experiments

For each user question, we prompt the model to generate a single valid DuckDB query without any explanations. The schema is always included in the prompt, and we also experiment with including database samples and descriptions. Below we describe the details of how the prompt was structured along with considerations that guided such formatting.

5.4.1 Schema

In line with previous approaches [TPC+24], [PLS+24], [CFL23], [LWZ+25], we represent the schema using its `CREATE table` statements. If the existing schema metadata specified explicit data types for columns and included notes on other constraints (such as primary/foreign keys or whether a column can be `NULL`), these details were also incorporated into the statement.

We do not perform schema filtering meaning; instead, we include all tables and columns from the relevant schema in the prompt for every query. It is not an issue for BIRD where there is only one assigned schema per database, but can be deemed as an oracle knowledge for the experiments with MDW-AMBIG as the MDW database spans 19 schemas, and it is impossible to know which ones are supposed to be used to answer the user question.

We made this decision because including all schemas in the prompt is infeasible: providing descriptions for all 2730 with an average length over 300 characters (true for all our experiments except the baseline description) and average token length equal to 4 [2] would result in 2730 * 300 / 4 = 204750 tokens, already exceeding the context window of Qwen. Furthermore, this approach would likely lead to many failures due to incorrectly selected tables or columns and overconvoluted context whereas in this study we want to concentrate more on evaluating SQL generation. Selecting the correct elements of the schema is indeed an integral part of the Text-to-SQL objective and remains partially tested in our setup, since we still include all the tables and columns within the relevant schema but not all of them are needed for the query. This way our context is around 50k tokens at maximum. This challenge with the extremely large schema of MDW highlights the relevance of schema linking in industrial setting, arguing with [MAJM24]. While this study focuses on SQL generation, applying a similar approach with leveraging a wider range of contextual metadata but now for schema filtering represents a promising research direction that remains unexplored here.

5.4.2 Samples & Descriptions

As mentioned above, we also experimented with adding the database samples to the prompt. When there are no privacy constraints, incorporating samples has proven highly beneficial for Text-to-SQL, particularly for questions requiring filtering by a specific value. Without knowing the exact value format, it can be easily mistyped—for example, writing `WHERE city = 'Krasnoyarsk'` when all values in the column are lowercase. Providing samples allows the model to infer the correct format.

We explored two approaches for integrating these samples into the prompt. The first was to append the sample rows directly below the schema. The second approach embedded the samples within the `CREATE TABLE` statements, placing example values after each column definition as was previously done in [TPC+24]. Preliminary runs revealed that the second approach consistently outperformed the first, yielding an average improvement of 2–3 percentage points: likely, the model finds it harder to associate samples placed at the end of the prompt with the corresponding columns defined earlier. Based on these results, we adopted the embedded format for incorporating samples into the schema.

In our experiments, samples were selected randomly, with the number of unique rows set to 3, following the approach of many existing systems [PLS+24], [GGL+25], [TPC+24]. To ensure the prompt remained within a reasonable length, each value was truncated to a maximum of 100 characters.

Similarly, the descriptions for each column were also embedded inside of schema. De-

[2] `https://ai.google.dev/gemini-api/docs/tokens?lang=python`

scriptions of tables were placed above the corresponding `CREATE` statement. A simplified example of the schema incorporating both samples and descriptions is demonstrated below:

```
Table devops.slo_overall

This table stores overall Service Level Objective (SLO) performance data,
tracking the success ratio over time.  Below is a corresponding DDL statement
with column descriptions and samples:

CREATE TABLE devops.slo_overall(

 ts_minute TIMESTAMP | This column stores the timestamp at a minute granularity.
It serves as the primary time key for recording Service Level Objective (SLO)
metrics.  | Sample values:  2025-04-24 04:34:00, 2025-04-27 08:31:00, 2025-05-01
10:09:00

 dt DATE | Stores the date for which the overall Service Level Objective (SLO)
metrics are recorded.  | Sample values:  2025-05-07, 2025-05-10, 2025-06-16

 success_ratio DOUBLE | Represents the ratio of successful operations or events,
primarily used to measure overall performance against Service Level Objectives
(SLOs).  | Sample values:  0.999435, 0.999655, 0.999307
);
```

Figure 5.8: Example prompt with descriptions and samples

5.4.3 Other

While representing the same language, SQL syntax can vary substantially across different dialects. For instance, the `strftime` function expects different argument orders in SQLite and DuckDB: SQLite takes the format string followed by the time string (e.g., `strftime('%s','2014-10-07 02:34:56')`), whereas DuckDB expects the time string first and then the format string, requiring an explicit data type casting (e.g., `strftime(DATE '1992-03-02', '%d/%m/%Y');`).

Although DuckDB has been rapidly gaining popularity in recent years, it is likely less present in the data used to train LLMs in comparison to other dialects, which could create a bias against DuckDB syntax. To mitigate this, we considered including a summary of DuckDB-specific SQL syntax in the prompt. Preliminary runs showed a slight benefit (within 1%) but we ultimately chose not to include it to avoid polluting the prompt context. The only exception was made for Qwen experiments, where we added a single line reminding the model to include the schema name when referencing tables as early experiments consistently failed due to this omission in every generated query.

We do not include in-context examples. While this could potentially improve performance, we chose to focus solely on leveraging contextual metadata rather than prompt-engineering techniques.

Chapter 6

Results

This chapter displays the Text-to-SQL performance of models given different contextual information.

6.1 BIRD results

We conduct experiments with and without database content samples, and with various types of descriptions added. Table 6.1 summarizes the results for BIRD-Dev.

Configuration	gemini2.5-flash	qwen2.5-32B	Input Token Count
Schema-only experiments			
schema	34.8%	20.2%	687
+ schema descriptions	32.2% (-2.6%)	18.2% (-2.0%)	3171
+ stats descriptions	35.4% (+0.6%)	18.4% (-1.8%)	3393
+ query annotation descriptions	38% (+3.2%)	19.4% (-0.8%)	5830
+ query patterns descriptions	36.8% (+2%)	18.4% (-1.8%)	6248
Schema + samples experiments			
schema + samples	42.2%	28.2%	3560
+ schema descriptions	42.4% (+0.2%)	20.6% (-7.6%)	6050
+ stats descriptions	42.6% (+0.4%)	21.2% (-7%)	6272
+ query annotation descriptions	43.8 % (+1.6%)	14.2% (-14%)	8696
+ query patterns descriptions	44% (+1.8%)	22.4% (-5.8%)	9127

Table 6.1: Performance (EX) Comparison by Configuration Type for BIRD-Dev. The number in brackets shows the difference between the baseline prompt settings for that experiment (w/ or w/o samples) and when different descriptions are added.

For the **Gemini 2.5 Flash** model, baseline and profiling descriptions offer little benefit

and can even notably degrade performance (-2.6% for baseline descriptions with no samples). The fluctuations of 0.2-0.6% translate into a delta of 1-3 items in the benchmark which can not be deemed a significant improvement.

More substantial performance increase is observed with the descriptions derived from query history (query annotations and frequent patterns) resulting in 2–3% performance boost for experiments with just a schema. This improvement also holds for the experiments including samples, though to a slightly lesser degree with 1.6-1.8% increase.

A closer examination of predictions accounting for a 2–3% change in the no-sample experiments reveals that many improvements come from value-related details in the column descriptions. For example, the description for the `bond.bond_type` column explicitly states that possible values are '=', '#' or '-'. Despite intentionally not including samples when generating the descriptions, it likely picks it up from the common expressions which are shown in Figure 6.1:

Generated description for `bond.bond_type`

Stores the type of chemical bond. This column is frequently used in WHERE clauses, often filtered by specific string values such as '=' (double bond), '#' (triple bond), and '-' (single bond) to identify bonds of a particular order. It commonly appears in WHERE clauses alongside molecule.label and atom.element to filter bonds based on the characteristics of the associated molecule or atoms. bond_type is also frequently selected, often together with bond_id.

Frequent expressions for `bond.bond_type`

Expression	Count
bond.bond_type = '='	10
bond.bond_type = '#'	9
bond.bond_type = '-'	7

Figure 6.1: Description derived from frequent expressions and corresponding expressions for `bond.bond_type` column

Without this information and when only schema is given, the model filters bond type by the value 'double', leading to incorrect results; the description eliminates this error. This issue can also be resolved by providing samples: indeed, in this example, the prediction is correct when samples are included.

Nonetheless, there are also cases where frequent expressions mentioned in the description contribute to improvements. In such cases, samples are not helpful, and these scenarios account for most of the observed prediction gains (1.6–1.8%) in the experiments with samples. A common pattern is when there has to be filtering by certain thresholds: e.g. some questions require filtering based on medical indicators such as abnormal levels of glutamic oxaloacetic transaminase (`Laboratory.GOT`) - or by the age group like teenager/elder for `users.Age`. Without knowing what exactly is a threshold for the abnormal GOT or how age groups are defined in this particular context, the model resorts to comparison with random values leading to an error. Query history contains filters that have been applied to these columns in the past, which the model can reuse in situations of uncertainty or when lacking prior domain knowledge. Often, it ends up being better

than a random guess, as demonstrated in some of our results described below.

For the abnormal GOT, the model incorrectly filters by `IS NOT NULL` or `> 0` when the correct threshold is `>60`, as indicated in the description derived from frequency patterns and shown in Figure 6.2.

Generated description for `Laboratory.GOT`

Represents the Glutamic Oxaloacetic Transaminase (GOT) level, a laboratory measurement typically indicating liver health. This BIGINT column is nullable, allowing for missing test results. It is primarily used in WHERE clauses, often in conjunction with patient.sex to filter results. A common usage pattern involves filtering for values less than 60, as seen in the expression GOT < 60.

Figure 6.2: Description for `Laboratory.GOT` column derived from query patterns

A similar idea is observed for `users.Age`: the query annotation description in Figure 6.3 mentions all age filters found in historical queries providing a clear signal to use `>65` as the boundary for the elderly group and preventing the model from filtering by `>60`, as it does when no descriptions are provided.

Generated description for `users.Age`

Stores the age of a user. It is commonly used in WHERE clauses to filter records by specific age values (e.g., 40) or ranges (e.g., 19 to 65, or greater than 65). The column is also utilized for counting users within particular age groups (e.g., 13-18) and can serve as a filter condition on joined results.

Figure 6.3: Description for `users.Age` column derived from query annotations

Interestingly, both of these are provided as evidence in BIRD (Figure 6.4) but we manage to recover them purely by analysing query history and without any domain knowledge.

For GOT:

Commonsense evidence:
Normal range: N < 60

For Age:

teenager: 13-18
adult: 19-65
elder: > 65

Figure 6.4: Evidence provided in BIRD benchmark

The erroneous cases where the model succeeds without descriptions but fails with them appear to be quite random. In some examples, it misinterprets the question, in others, it omits required output columns or forgets certain filters mentioned in the question. A possible explanation is that with a larger context, the model's attention spreads out, occasionally causing it to lose key details from the question.

For **Qwen2.5-32B**, the results are more straightforward: adding descriptions does not help. The number of execution errors for the predicted queries significantly increased with longer context, with most errors being caused by column/table mix-ups or hallucinating non-existent schema elements. This suggests that simpler models do not handle long context well so enriching it with the descriptions is not only unhelpful but rather detrimental to performance.

6.2 MDW-AMBIG

A similar trend to that observed for BIRD can be seen in the MDW results, presented in Table 6.2. Given the dataset size of only 50 samples, each 2% improvement corresponds to a single example.

Configuration	gemini2.5-flash	qwen2.5-32B	Input Token Count
Schema-only experiments			
schema	36%	18%	2722
+ schema descriptions	40% (+4%)	16% (-2%)	10610
+ stats descriptions	46% (+10%)	8% (-10%)	30395
+ query annotation descriptions	44% (+8%)	10% (-8%)	26329
+ query patterns descriptions	52% (+16%)	8% (-10%)	26079
Schema + samples experiments			
schema + samples	42%	12%	20107
+ schema descriptions	40% (-2%)	8% (-4%)	28089
+ stats descriptions	48% (+6%)	10% (-2%)	47491
+ query annotation descriptions	46% (+4%)	10% (-2%)	43649
+ query patterns descriptions	52% (+10%)	4% (-8%)	43411

Table 6.2: Performance (EX) Comparison by Configuration Type for MDW-AMBIG. The number in brackets shows the difference between the baseline prompt settings for that experiment (w/ or w/o samples) and when different descriptions are added.

As with BIRD, introducing the descriptions improves scores for **Gemini 2.5-Flash** experiments. The most effective descriptions are the ones based on frequent patterns. The error review reveals some inexplicable cases but there are also a few clear demonstrations of how the predictions benefit from the descriptions.

Similarly to BIRD, the improvement partially comes from following the expressions mentioned in descriptions. The difference, however, is that in the MDW dataset the expressions are more complex. The model predicts them correctly by directly copying it from the descriptions, and without this information, current models stand less chance of success, either due to the unclearness of the question or the inherent complexity of the expression.

For instance, the model gets the quite tricky expression regarding the revenue coming from certain types of services shown in Example 6.5. In other predictions it confuses columns, values or even tables as well as the order of filtering and aggregation (though here it does not make a difference, it can be critical in other cases).

Question: *Can you provide a report listing every id for clients that had any revenue activity on June 24th, 2025, and include a column for each showing their total earnings specifically from storage services?*

Schema + frequent patterns description:

The item column specifies the particular product or service that generated revenue. It is frequently used to categorize and analyze revenue, enabling users to segment total revenue into distinct components such as 'Compute Unit hours', 'Storage (GB-days)', and 'Platform Fee'. This segmentation is commonly achieved through `SUM(revenue) FILTER (WHERE item IN (...))` expressions, which are often aliased as compute, storage, or platform for clarity in analytical queries...

Correct prediction

Schema + Query Pattern descr:

```
SELECT
 customer_id,
SUM(revenue) FILTER (
 WHERE item = 'Storage (GB-days)'
) AS total_storage_earnings
FROM orb.revenue_by_day
WHERE
 dt = '2025-06-24'
GROUP BY
 customer_id;
```

Examples of incorrect predictions:

Schema + Query Annotations descr

```
SELECT
 customer_id,
 SUM(revenue)
 AS total_earnings_from_storage_services
FROM orb.revenue\_by\_day
WHERE
 dt = '2025-06-24' AND item = 'storage'
GROUP BY
 customer_id;
```

Schema + samples + Schema descr:

```
SELECT
customer_id,
SUM(CASE WHEN item_category = 'Storage'
THEN gross_revenue ELSE 0 END)
AS total_storage_earnings
FROM orb.gross_revenue_detailed
WHERE
 revenue_date = '2025-06-24'
GROUP BY
 customer_id;
```

Figure 6.5: Example of correct SQL query prediction with query pattern descriptions vs incorrect predictions.

While frequent pattern information was beneficial for MDW, descriptions from annotated queries proved less helpful than in BIRD experiments. This could be because they mostly describe patterns in text rather than including expressions directly (see the comparison in Example 6.3), making it difficult to reconstruct the exact expressions when they are more complex. In contrast, BIRD's expressions are generally simpler, as discussed in the reference comparison section 5.1.3, and can be reconstructed more easily. In this case,

the model uses `FILTER` with the description on the left and produces the correct result. Although the query annotation description on the right also contains `FILTER`, the model's predictions ignore it.

Frequent pattern description:	**Query Annotation description:**
`The item column specifies the particular product or service that generated revenue. It is frequently used to categorize and analyze revenue, enabling users to segment total revenue into distinct components such as 'Compute Unit hours', 'Storage (GB-days)', and 'Platform Fee'. This segmentation is commonly achieved through SUM(revenue) FILTER (WHERE item IN (...)) expressions, which are often aliased as compute, storage, or platform for clarity in analytical queries...`	`Represents the name or description of the specific service or product item for which revenue was generated. This column is frequently used in FILTER clauses, particularly within SUM aggregate functions, to categorize and aggregate revenue by distinct service types such as 'compute', 'storage', and 'platform'. It enables a detailed breakdown of total revenue into its constituent service components...`

Table 6.3: Comparison of query pattern and query annotation descriptions for column `item`

Another interesting example is when the right table is selected potentially because of relative frequency mentioned in table descriptions. When the question is about generated revenue, there are a few similarly named candidate tables (the italic text marks the parts of the table descriptions that were derived from frequent patterns):

- `gross_revenue_detailed` - *"...This table **has not been used** in any SQL queries in the past..."*
- `gross_revenue` - *"...It is **less frequently** queried compared to other tables..."*
- `revenue_by_day` - *"...This table is **frequently** used in queries and commonly joined..."*

The table referenced in the ground truth queries is always `revenue_by_day`, the most frequently used one. It is selected almost exclusively in experiments involving query pattern descriptions where this is explicitly stated. It is hard to confidently claim that this particular factor was the decisive one for the table selection but it is suggested by how tables were selected in other experiments (see Figure 6.6): the 'never-used' table was selected consistently as often as the 'frequently used' one. For `MDW`, table and column selection is more critical than for `BIRD` due to name similarity, making any signal, such as this one, potentially relevant, especially since the same data tends to be queried repeatedly.

Importantly, the performance improvement gained from adding samples is not as substantial and can be overshadowed by the addition of descriptions, something we did not

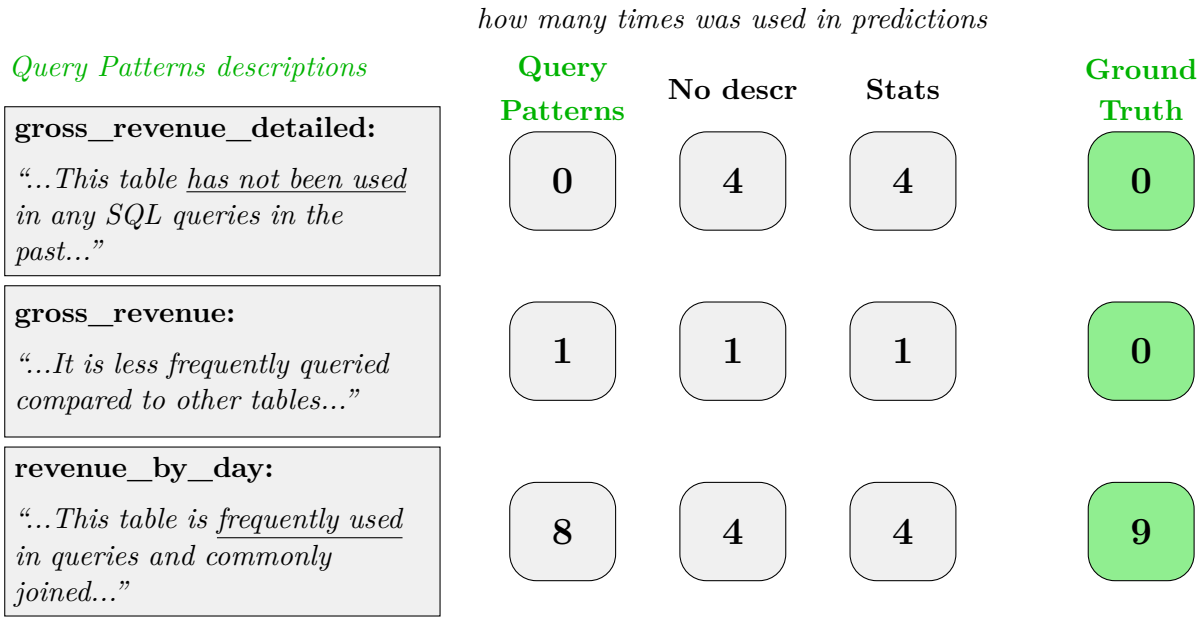

Figure 6.6: Table descriptions derived from query patterns and count of times when the table was used in a prediction given different descriptions. The count of table references in the corresponding ground truth queries is shown in green.

observe with BIRD. In fact, including descriptions without samples yields better results in the majority of experiments than using samples without descriptions. It highlights the lesser importance of samples in MDW experiments rather than BIRD experiments, which can be linked to a lack of filters involving synonymous or interchangeable database values in MDW-AMBIG queries.)

Finally, one can discover a huge difference in context length: 3-6k context for `BIRD` to over 10k input tokens for `MDW`, all because of the schema sizes. The large schema issue becomes considerably more pronounced when dealing with the real-world database. It becomes a killer factor for **Qwen**: on average half of the predictions are not executable because of misspelling or hallucinating schema identifiers (like `billing_data` to real `billing_details`).

Chapter 7

Discussion

In this chapter we compare the experimental results across different settings, discuss the advantages and disadvantages of our proposed method for generating and applying schema description and outline potential directions for its improvement and broader application.

7.1 Overall performance: How useful are query-log-informed descriptions for LLMs in SQL generation?

Extending semantic context impacts different models disparately. For **Qwen2.5-32B**, the weaker LLM, quality significantly deteriorated across all tested settings when descriptions were included, likely because it struggles to process longer contexts effectively.

In experiments with **Gemini 2.5-Flash**, adding descriptions to the prompt increased scores, both with and without database samples. The margin varied across datasets. The difference largely arose from variations in dataset sizes (each item in `MDW` accounts for the 2% of a total performance while only 0.2% for `BIRD`) and the Text-to-SQL challenges covered by the questions. In `BIRD` the improvement mainly stemmed from some of match-based and reasoning questions which constitute only a portion of the dataset. On the other side, `MDW` by design is more concentrated around ambiguous questions and identifier names, where we expected the descriptions to be beneficial. However, these scenarios are very common in the industrial setting, so, even while our experiments are an approximation of the real-world environment, the improvement on those is tangible and promising.

In experiments on both datasets, query history information proved more useful than profiling details or raw name-based descriptions. The greatest benefit came from **specific expressions** mentioned in the descriptions - they either helped to recover the contextual knowledge from previous filters and reusing them, or to construct complex expressions that are difficult to reproduce just from a natural question without a direct example. For more diverse expressions in MDW, descriptions derived from query log patterns were

particularly helpful, as they allowed direct copying, whereas query annotation descriptions, being more textual, were less effective in such cases. This could be mitigated with an improved prompting engineering, though even then there remains a risk of overlooking details due to sampling. Moreover, this method does not enable **relative comparisons of pattern frequencies**, which can sometimes be beneficial.

Co-occurrence information did not show clear utility for the SQL generation so it can potentially be excluded from the prompt to not occupy further context. It still, however, has potential for aiding schema linking. Notes on **primary-foreign key relationships** also appeared not as useful as anticipated: they seem to be typically well inferred from names and values alone, without requiring additional specification.

7.2 Comparing to the official BIRD leaderboard

While our results are substantially lower than those on the official `BIRD` leaderboard,[1] several factors should be considered.

First, we use `BIRD-Mini-Dev`[2] which contains more moderate items in comparison to the prevalence of simple questions in the standard `BIRD-Dev`.

Secondly, we perform the translation to `DuckDB` SQL dialect, the more recent one and likely less represented in the LLM's training data compared to `SQLite3` which is the main dialect of the `BIRD` benchmark. Despite explicitly requesting `DuckDB` queries in the prompt, there are plenty of syntax errors in the predictions where `SQLite3` functions are used instead of `DuckDB` alternatives contributing to lower scores.

Thirdly, we are not utilizing the evidence knowledge provided by the benchmark which is concise and verified by the authors. Instead we attempt to recover this information from raw queries, imitating real company environments where such evidence is typically not provided.

Finally, the method introduced in this study is more focused and addresses a particular type of challenge. If maximizing an overall performance is the primary goal, it would be more effective to first consider other commonly present Text-to-SQL modules such as self-consistency with multiple candidates and self-revision. Given the instability of LLM predictions, these modules would be most effective in achieving better benchmark performance.

[1] `https://bird-bench.github.io/`

[2] `https://github.com/bird-bench/mini_dev?tab=readme-ov-file#ex-evaluation`

7.3 Benefits and Limitations of query-logs-informed descriptions

In addition to the Text-to-SQL benchmark results, we believe it is worth also discussing the broader pros and cons of our method.

7.3.1 Advantages

Beyond the overall positive effect of incorporating the descriptions into Text-to-SQL context, the proposed approach offers several additional benefits.

One of the main advantages of this method is low inference latency. Generating the descriptions requires some time, but it can be done offline, so, once generated, they can be easily added to the prompt, keeping the Text-to-SQL within one LLM call and incurring minimal computational overhead. This makes this method especially practical for online SQL assistants where the inference speed matters a lot in the user experience. It is also highly efficient in comparison to the other existing solutions which include generating multiple candidates and performing additional reasoning steps, which can result sometimes in over a hundred real-time LLM calls. Moreover, context enhancement with the descriptions can also easily be combined with such methods, potentially leading to even better performance.

In addition to SQL translation, the generated descriptions and statistics are very versatile. They can facilitate data exploration (for example, through Web Interface), support other AI applications and data management tools, and provide valuable insights for data documentation. For instance, despite less useful in our Text-to-SQL experiments, we can recover the relationships between tables with primary and foreign keys, which is essential for understanding the logic of how the data is structured. Figure 7.1 displays an example of relationships schema for codebase_comumunity database from BIRD.

As can bee seen from the example, query history has not captured all of documented relationships but helped to discover a few of new ones. It even helped to detect an error in the benchmark where the same table was unintentionally referenced twice in a join (green dashed line). This suggests that such an analysis can be a promising automatic method for recovering or verifying documentation.

7.3.2 Disadvantages

On the negative side, such a method is not very effective if schema and identifiers are simple and well-interpretable as it targets more specific subset of cases with ambiguous names, relying on domain knowledge and in scenarios when there are a lot repetitive patterns involved in the queries.

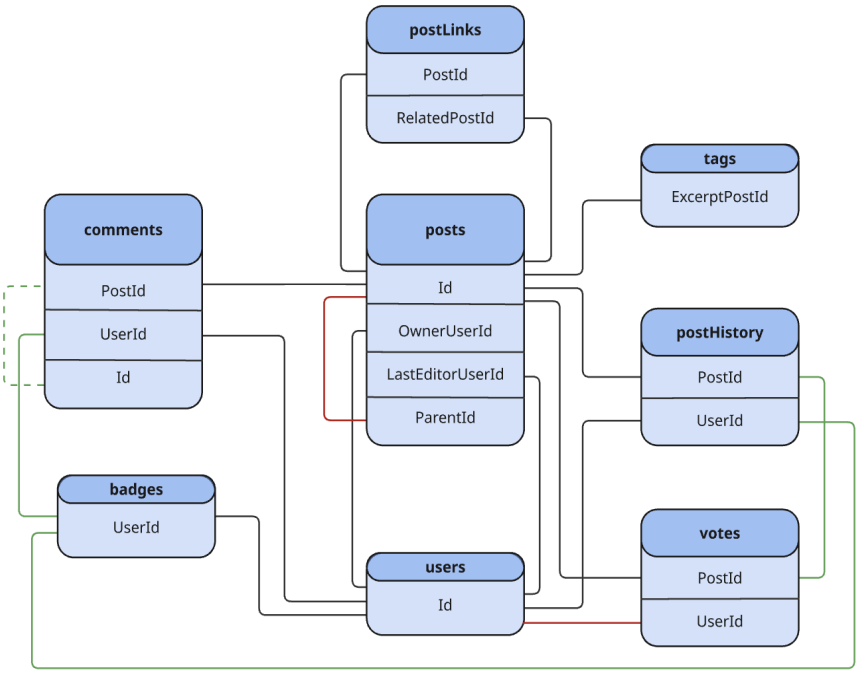

Figure 7.1: Documented and recovered relationships for `codebase_community` database from BIRD. Highlighted with red are the relationships that were not discovered through the query logs but are present in the documentation while the green connections show the relationships missing in documentation but revealed through analyzing the historical queries. The black relationships were revealed by both.

Moreover, our study focuses mainly on query history as the knowledge source which has two major drawbacks: firstly, historical queries may not be present at all or cover very narrow set of data, and, secondly, is not always trustworthy. The undercoverage of the data is quite realistic and is partly shown in Figure 5.4, where 30% to 40% of the columns remain unused in the train sets for BIRD and MDW. However, the generated descriptions still yield a performance gain in Text-to-SQL, even with limited coverage, due to again the repetitive nature of queries. Even with not a large training set (only 1039 queries for 806 columns in BIRD), it still proves beneficial as long as the usage of the identifiers is consistent. It is, however, still a problem for other potential applications targeting user-facing data understanding: we cannot provide any assistance for the identifiers with no information from query history.

Regarding the second issue, query logs indeed do not guarantee their correctness, thus, the extracted information could be misleading rather than helpful. Furthermore, it is common for users to iteratively refine and re-run the same query (this usage pattern is visible in the clustering of queries in 5.6), which can significantly skew the statistics, also

making them less reliable. This highlights a need for a proper ways to pre-filter query history before extracting the insides from it.

Due to these issues and similarly to other processes involving LLMs, a human-in-the-loop review seems necessary which in turn also adds to the required effort. It is reinforced by a need to maintain the description which over time can be challenging. The dynamic nature of data warehouse schemas means that new tables and columns are added, existing ones are modified or removed, and historical queries may reference outdated elements or redefined expressions. This task is complicated by the fact that descriptions contain a lot inter-references when one descriptions mentions other columns/tables, and if one of those is altered, the descriptions for all the related identifiers also must be updated. While generating the descriptions for all identifiers in the database is rather cheap (according to our calculations, it costs 0.5$ for MDW - check the Appendix A.2 for more details), it still requires considerable amount time and some engineering effort to identify the scenarios when this regenerations has to be triggered. Without proper maintenance, such documentation can become useless or even detrimental to performance and data understanding.

7.4 Future Work

In this section we discuss ideas for potential studies that can build on top of research.

As demonstrated by some of our case studies, query logs analysis can provide relevant signals for schema filtering. As shown in Example 6.6, knowing how often a particular table has been used historically can hint which column to select during SQL generation, but can be applied even earlier when constructing the prompt. Given that it is the subsets parts of the data that are usually getting utilized, this can often prove accurate. For the same reason, knowing that some columns or tables often co-occur together might appear beneficial as well: for instance, if there is a certain expression-formula, but in calculations it uses some seemingly unrelated identifiers. Moreover, descriptions or some other form of query history analysis can add new features to the search space if schema retrieval is done via vector search.

Another interesting direction to investigate is analyzing query history on the level of CTEs and subqueries. In long analytical queries, not just tables and columns but entire CTEs often serve as individual semantically loaded building blocks: they usually have meaningful names and are reused in the identical form across different queries. When conducting analytics, users treat them as standalone and complete conceptual units which can be stored and analyzed similarly to expressions and individual column/table occurrences.

Building upon the previous ideas, a natural progression would be moving from textual form to a more structured representation, potentially forming a foundation for a semantic layer - a bridge between business users and raw data. The descriptions in such a layer

can facilitate data discovery and help users understand the data. At the same time, analyzing query history can serve as a mechanism for the layer's automatic enrichment: the revealed expressions and CTEs that reflect business metrics can be identified, stored and reused to ensure consistency across the organization. One idea is to represent the layer as a knowledge graph with hierarchical organization (similarly to [AYZ+25]): the minimal units are columns and tables which can relate to other (co-occur in queries together, connected via primary-foreign key relationships, or share the same lineage) and combine to form a higher-level components such as expressions, then CTEs and subqueries, and then ultimately complete business concepts.

Finally, this study reveals critical discrepancies between BIRD, the most commonly used Text-to-SQL benchmark, and real-world industrial settings. Current benchmarks like BIRD, while widely adopted for evaluation, fail to capture the complexity and constraints of industrial databases, creating a disconnect where high-performing models on benchmarks become ineffective or unusable in actual production settings. This calls for a new benchmark that would test SQL generation in conditions, more closely imitating the real industry, and consider the constraints stemming from it. We highlight three key considerations:

- **Schema complexity needs to be more realistic**: The schema should be large and contain similar, confusing or ambiguous names, and the join keys should not always be obvious;
- **Query history should be available and capture how users write queries**: Particularly their repetitive patterns, as this is a fundamental characteristic of real-world usage, and systems should be tested on their ability to effectively leverage these patterns;
- **Latency and query optimization should matter**: Performance metrics should at least be measured and reported as part of the evaluation.

Such a benchmark would provide a realistic assessment not of how well existing approaches can generate SQL in general, but of how they perform with production-level databases in industrial environments.

Chapter 8

Conclusions

In this study, we address the problem of lacking or low-quality documentation for database schemas elements. Combined with the complexity of these schemas, it poses challenges for both users intending to explore and analyze their data as well as AI agents attempting to interact with the data. To mitigate this problem, we generate informative textual descriptions for schema elements based on query history that can facilitate multiple exploratory applications and provide context for AI features. We then assess the utility of these descriptions serving as context for SQL generation, one of the most relevant applications.

First, we propose two methods to answer RQ1.

- **RQ1:** How can informative descriptions for schema elements be automatically synthesized from existing query history?

Both of the methods use an LLM to produce the final description; however, the information presented in the prompt differs between the approaches. The first approach relies heavily on LLMs and involves annotating a random sample of queries with LLMs, which are then combined into a unified list. The second method employs query pattern mining: parsing each query from the history and extracting all column and table references as well as expression, which are then aggregated to form a usage profile reporting how frequently a certain identifier was used, in which clauses, with which other identifiers, and as part of which expressions.

The resulting descriptions were compared with those generated from raw schema alone and a combination of raw schema and column statistics. We specifically test the effectiveness of adding these descriptions in a context for Text-to-SQL, which brings us to RQ2:

- **RQ2:** To what extent are such descriptions beneficial for LLMs in the context of SQL generation?

For our evaluation, we used two different datasets: a traditional Text-to-SQL benchmark, widely adopted in research, and a manually collected set of queries that were ran by real user against the production-level database. The overall impact of incorporating the descriptions derived from query history is positive but it varies across datasets and experiments. The improvement over other descriptions or no descriptions ranges from 1.5 to 3% for the research benchmark, and reaches up to 10% for the production dataset, indicating higher usefulness in real world scenarios. This is supported by our observations that the benefit from descriptions largely stems from the repetitive nature of how real users write queries. As a result, information from query logs can become extremely helpful: the model can retrieve prominent complex expressions from descriptions, which is more reliable than generating it from scratch, or select the identifiers based on how frequently they were used previously, which is likely to be correct.

While the proposed method has certain limitations mainly tied to the quality of query logs and necessity to verify and maintain the descriptions, it also demonstrates versatility with potential to serve as valuable contextual enhancement for any AI applications dealing with structured data. Furthermore, it opens promising directions for applications involving semantic analysis on top of data warehouses and more robust integration of LLMs into enterprise data ecosystems.

Bibliography

[Ano24] Anonymous. Tabmeta: Table metadata generation with LLM-curated dataset and LLM-judges. In *Submitted to ACL Rolling Review - June 2024*, 2024. under review.

[AYZ+25] Qi An, Chihua Ying, Yuqing Zhu, Yihao Xu, Manwei Zhang, and Jianmin Wang. Ledd: Large language model-empowered data discovery in data lakes. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD '25)*, Berlin, Germany, 2025. ACM.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[CFL23] Shuaichen Chang and Eric Fosler-Lussier. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings, 2023.

[CHS04] Philipp Cimiano, Siegfried Handschuh, and Steffen Staab. Towards the self-annotating web. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, page 462–471, New York, NY, USA, 2004. Association for Computing Machinery.

[Col24] Collibra. Mastering the art of data intelligence: Empowering collibra with chatgpt, 2024. Accessed: 2025-04-24.

[Col25] Collibra. Ai generative descriptions factsheet, 2025. Accessed: 2025-04-24.

[CYXH21] Ruichu Cai, Jinjie Yuan, Boyan Xu, and Zhifeng Hao. SADGA: Structure-aware dual graph aggregation network for text-to-SQL. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[Dat24a] Databricks Inc. How we improved databricksiq llm quality: Ai-generated table comments, 2024. Accessed: 2025-04-24.

[Dat24b] DataHub Team. Ai-assisted data catalogs: Llm + knowledge graphs guide. `https://datahub.com/blog/ai-assisted-data-catalogs-an-llm-powered-by-knowledge-graphs-for-metadata-disc` 2024. Accessed: 2025-08-21.

[Dat25a] Databricks Inc. Ai-generated comments in unity catalog, 2025. Accessed: 2025-04-24.

[Dat25b] Dataedo. Ai autodocumentation, 2025. Accessed: 2025-04-24.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. cite arxiv:1810.04805Comment: 13 pages.

[DCW03] Alexiei Dingli, Fabio Ciravegna, and Yorick Wilks. Automatic semantic annotation using unsupervised information extraction and integration. In *K-CAP 2003 Workshop on Knowledge Markup and Semantic Annotation*, pages 1–8, Sanibel, Florida, USA, 2003. CEUR-WS.org. Accessed: 2025-04-24.

[DGL+23] Longxu Dou, Yan Gao, Xuqi Liu, Mingyang Pan, Dingzirui Wang, Wanxiang Che, Dechen Zhan, Min-Yen Kan, and Jian-Guang Lou. Towards knowledge-intensive text-to-sql semantic parsing with formulaic knowledge, 2023.

[dHSW02] erik duval, Wayne Hodgins, Stuart Sutton, and Stuart Weibel. Metadata principles and practicalities. *D-Lib Magazine*, 8:–, 04 2002.

[DRX+25] Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration, 2025.

[DSL+20] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. Turl: Table understanding through representation learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1949–1964. Association for Computing Machinery, 2020.

[EMSS00] M. Erdmann, A. Maedche, H.-P. Schnurr, and S. Staab. From manual to semi-automatic semantic annotation: about ontology-based text annotation tools. In *Proceedings of the COLING-2000 Workshop on Semantic Annotation and Intelligent Content*, COLING '00, page 79–85, USA, 2000. Association for Computational Linguistics.

[Eur19] European Commission. Dcat application profile for data portals in europe (dcat-ap). `https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/dcat-application-profile-data-portals-europe`, 2019. Accessed: 2025-04-24.

[FXT+24] Xi Fang, Weijie Xu, Fiona Anting Tan, Ziqing Hu, Jiani Zhang, Yanjun Qi, Srinivasan H. Sengamedu, and Christos Faloutsos. Large language models (LLMs) on tabular data: Prediction, generation, and understanding - a survey. *Transactions on Machine Learning Research*, 2024.

[GGL+25] Satya Krishna Gorti, Ilan Gofman, Zhaoyan Liu, Jiapeng Wu, Noël Vouitsis, Guangwei Yu, Jesse C. Cresswell, and Rasa Hosseinzadeh. Msc-sql: Multi-sample critiquing small language models for text-to-sql translation. In *The 2025 Annual Conference of the Nations of the Americas Chapter of the ACL*, 2025.

[GL25] Yingqi Gao and Zhiling Luo. Automatic database description generation for text-to-sql. 2025.

[GLL+24] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv preprint arXiv:2411.08599*, 2024.

[GTT+23] Chunxi Guo, Zhiliang Tian, Jintao Tang, Pancheng Wang, Zhihua Wen, Kang Yang, and Ting Wang. A case-based reasoning framework for adaptive prompting in cross-domain text-to-sql. *CoRR*, abs/2304.13301, 2023.

[GWL+23] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR*, abs/2308.15363, 2023.

[GXG+24] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[HDW23] Zezhou Huang, Pavan Kalyan Damalapati, and Eugene Wu. Data ambiguity strikes back: How documentation improves gpt's text-to-sql, 2023.

[HNM+20] Jonathan Herzig, Pawel Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. Tapas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4320–4333. Association for Computational Linguistics, 2020.

[HW24] Zezhou Huang and Eugene Wu. Cocoon: Semantic table profiling using large language models. In *Proceedings of the 2024 Workshop on Human-In-the-Loop Data Analytics*, HILDA 24, page 1–7, New York, NY, USA, 2024. Association for Computing Machinery.

[HZZ+23] Xinyi He, Mengyu Zhou, Mingjie Zhou, Jialiang Xu, Xiao Lv, Tianle Li, Yijia Shao, Shi Han, Zejian Yuan, and Dongmei Zhang. Anameta: A table understanding dataset of field metadata knowledge shared by multi-dimensional data analysis tasks, 2023.

[LHQ+24] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[LLC+24] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. The dawn of natural language to sql: Are we fully ready? *Proc. VLDB Endow.*, 17(11):3318–3331, July 2024.

[LPKP24] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation, 2024.

[LPR21] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. Kaggledbqa: Realistic evaluation of text-to-sql parsers, 2021.

[LSL+25] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A survey of nl2sql with large language models: Where are we, and where are we going?, 2025.

[LSX20] Xi Victoria Lin, Richard Socher, and Caiming Xiong. Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In Trevor Cohn,

Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4870–4888, Online, November 2020. Association for Computational Linguistics.

[LWZ+24] Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Rui Zhao, et al. Pet-sql: A prompt-enhanced two-stage text-to-sql framework with cross-consistency. *arXiv preprint arXiv:2403.09732*, 2024.

[LWZ+25] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. Omnisql: Synthesizing high-quality text-to-sql data at scale, 2025.

[LYKK22] Jinhyuk Lee, Wonjin Yoon, Donghyeon Kim, and Jaewoo Kang. Annotating columns with pre-trained language models. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*, pages 1234–1246. Association for Computing Machinery, 2022.

[LZL+24] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql, 2024.

[LZLC23] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *AAAI*, 2023.

[MAJM24] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. The death of schema linking? text-to-sql in the age of well-reasoned language models, 2024.

[MLG23] Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models, 2023.

[MW23] Zhengjie Miao and Jin Wang. Watchog: A light-weight contrastive learning based framework for column annotation. *Proc. ACM Manag. Data*, 1(4), December 2023.

[MZX+25] Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. Sql-r1: Training natural language to sql reasoning model by reinforcement learning, 2025.

[Not24] Notion. A brief history of notion's data catalog. `https://www.notion.com/blog/a-brief-history-of-notions-data-catalog`, 2024. Accessed: 2025-01-27.

[PLS+24] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql, 2024.

[PR23] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*, 2023.

[PR24] Mohammadreza Pourreza and Davood Rafiei. Dts-sql: Decomposed text-to-sql with small large language models, 2024.

[PTS+25] Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, and Sercan "O. Arik. Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql, 2025.

[RBS09] Marko A. Rodriguez, Johan Bollen, and Herbert Van De Sompel. Automatic metadata generation using associative networks. *ACM Trans. Inf. Syst.*, 27(2), March 2009.

[SAM+24] Ruoxi Sun, Sercan Ö. Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. Sql-palm: Improved large language model adaptation for text-to-sql (extended), 2024.

[SBME24] Eitam Sheetrit, Menachem Brief, Moshik Mishaeli, and Oren Elisha. Rematch: Retrieval enhanced schema matching with llms, 2024.

[SKDK25] Mayank Singh, Abhijeet Kumar, Sasidhar Donaparthi, and Gayatri Karambelkar. Leveraging retrieval augmented generative llms for automated metadata description generation to enhance data catalogs, 2025.

[Sno25a] Snowflake Inc. Generate descriptions with cortex, 2025. Accessed: 2025-04-24.

[Sno25b] Snowflake Inc. Generate descriptions with cortex: Cost considerations, 2025. Accessed: 2025-04-24.

[SSJG25] Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. Automatic metadata extraction for text-to-sql, 2025.

[SZZ+23] Yuan Sui, Jiaru Zou, Mengyu Zhou, Xinyi He, Lun Du, Shi Han, and Dongmei Zhang. Tap4llm: Table provider on sampling, augmenting, and packing semi-

structured data for large language model reasoning. *Conference on Empirical Methods in Natural Language Processing*, 2023.

[TPC+24] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[VSP+23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[W3C20] W3C. Data catalog vocabulary (dcat) – version 2. `https://www.w3.org/TR/vocab-dcat-2/`, 2020. W3C Recommendation.

[War03] J. Ward. A quantitative analysis of unqualified dublin core metadata element set usage within data providers registered with the open archives initiative. In *2003 Joint Conference on Digital Libraries, 2003. Proceedings.*, pages 315–317, 2003.

[WHL+24] Niklas Wretblad, Oskar Holmström, Erik Larsson, Axel Wiksäter, Oscar Söderlund, Hjalmar Öhman, Ture Pontén, Martin Forsberg, Martin Sörme, and Fredrik Heintz. Synthetic sql column descriptions and their impact on text-to-sql performance, 2024.

[WKLW98] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Rfc2413: Dublin core metadata for resource discovery, 1998.

[WRY+25] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, 2025.

[WSL+20] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. pages 7567–7578, 01 2020.

[WSL+21] Daheng Wang, Prashant Shiralkar, Colin Lockard, Binxuan Huang, Xin Luna Dong, and Meng Jiang. TCN: table convolutional network for web table interpretation. In *WWW*, pages 4020–4032, 2021.

[WWS+23] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

[XXZG25] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment, 2025.

[YGU+22] Jingfeng Yang, Aditya Gupta, Shyam Upadhyay, Luheng He, Rahul Goel, and Shachi Paul. Tableformer: Robust transformer modeling for table-text encoding. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 528–537. Association for Computational Linguistics, 2022.

[YL05] Hsin-Chang Yang and Chung-Hong Lee. Automatic metadata generation fo rweb pages using a text mining approach. In *International Workshop on Challenges in Web Information Retrieval and Integration*, pages 186–194, 2005.

[YNYR20] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426. Association for Computational Linguistics, 2020.

[YNYR21] Da Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. Tabbie: Pretrained representations of tabular data. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3446–3456. Association for Computational Linguistics, 2021.

[YZY+19] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2019.

[ZSS+23] Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Shen Wang, Huzefa Rangwala, and George Karypis. NameGuess: Column name expansion

for tabular data. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13276–13290, Singapore, December 2023. Association for Computational Linguistics.

Appendix A

Generation Prompts

This appendix contains the prompts used for generating table and column descriptions.

A.0.1 System Prompt

All description generation prompts use the following system prompt:

```
You are a helpful assistant that can generate descriptions for columns
and tables based on the user input to give users an easier time
understanding the data and writing SQL queries.
```

A.0.2 Schema, stats and Query Annotation Description Prompt for Columns

```
### Task - Generate a precise description for the {{ column }} column in the
{{ table }} table.
Your description should include:
- What is this column and what is its primary purpose;
- Additional useful information formatted as a new sentence;
If there is not enough information to make a valid prediction, return empty
string.

### Requirements
- Focus solely on confirmed details, try to include the most relevant
information
- Keep the description concise and factual, don't use introductory phrases
like "The purpose of the column is...".  Go straight to the point
- Exclude any speculative or additional commentary
- DO NOT return the phrase "in the {{ table }} table " in your description.

DO NOT return anything else except the generated column description.  This
is very important.  The answer should be only the generated text aimed at
describing the column.

### Additional information
Here are the schema details of the {{ table }} table:
{{ schema }}
```

A.0.3 Schema, stats and Query Annotation for Description Prompt Tables

```
### Task - Generate a precise description for the {{ table }} table.
Your description should include:
- What is this table and what is its primary purpose;
- Additional useful information formatted as a new sentence;
If there is not enough information to make a valid prediction, return empty
string.

### Requirements
- Focus solely on confirmed details, try to include the most relevant
information
- Keep the description concise and factual, don't use introductory phrases
like "The purpose of the table is...".  Go straight to the point
- Exclude any speculative or additional commentary

DO NOT return anything else except the generated table description.  This
is very important.  The answer should be only the generated text aimed at
describing the table.

### Additional information
Here are the schema details of the {{ table }} table:
{{ schema }}
```

A.0.4 Query Patterns Description Prompt

```
### Task - Generate a precise description for the {{ column }} column in the
{{ table }} table.
Your description should include:
- Primary purpose of the column;
- Additional useful information formatted as a new sentence;
- Relevant information on how the column is used in queries if any;

### Requirements
- Focus solely on confirmed details
- Keep the description concise and factual, don't use introductory phrases
like "The purpose of the column is...".  Go straight to the point
- While being precise, include as much valuable information as possible into
the description.  Try to provide valuable insights into common or meaningful
usage patterns which would be useful for a user to know when writing SQL
queries.  This can include specifying valuable expressions or more details
about usage in certain clause types
- If there is a lot of potentially valuable info, feel free to extend the
description
- Don't mention particular numbers of queries or z-scores
- Information about columns used in JOIN conditions is very useful
- If the column is not often used in queries, it can be also useful to know
- Include particular expressions into a description if they have a particular
business meaning (it can be very nice to know!).  If possible, you can also
describe what this expression means in a business context
- Expression aliases can a very good signal about business value of certain
expression.  This can indicate that the expression is used for a particular
purpose.  If so, it is probably worth including the expression in the
description
- Exclude any speculative or additional commentary
- DO NOT return the phrase "in the {{ table }} table " in your description.

DO NOT return anything else except the generated column description.  This
is very important.  The answer should be only the generated text aimed at
describing the column.

### Additional information
Here are the schema details of the {{ table }} table:
{{ schema }}
```

A.0.5 Query Patterns Table Description Prompt

```
### Task - Generate a precise description for the {{ table }}.
Your description should include :
- Primary purpose of the table
- Additional useful information (if apparent from the schema), formatted as a
new sentence
- Relevant information on how the table is used in queries if any;
If no useful information is available or if the name and the details in the
schema do not suffice to ascertain purpose and useful details, return "".

### Requirements
- Focus solely on confirmed details from the provided schema and additional
information
- Keep the description concise and factual; don't use introductory phrases
like "The purpose of the table is...".  Go straight to the point
- Try to include as much confirmed information as possible which would be
useful for a user to know when writing SQL queries
- Don't mention particular numbers of queries or z-scores
- Please don't simply list the columns in the description, it is not useful -
try to analyze it and extract their business meaning
- Information about potentially joinable tables is important
- If the table is not often used in queries, it can be also useful to know
- You can include specific expressions as they are in the description if they
carry business relevance or are useful for the user to understand.
- Exclude any speculative or additional commentary

DO NOT return anything else except the generated table description.  This
is very important.  The answer should be only the generated text aimed at
describing the table.

### Additional information
Here are the schema details of the {{ table }} table:
{{ schema }}
```

A.1 SQL Generation Prompts

The prompt used for SQL generation experiments:

A.1.1 System Prompt

```
You are a helpful assistant that can generate DuckDB / MotherDuck SQL
queries based on the user input.  You do not respond with any human
readable text, only SQL code.
```

A.1.2 User Prompt Template for SQL Generation

```
Output a single DuckDB SQL query without any explanation and do
not add anything to the query that was not part of the question.

Here is the database schema overview:
{{schema}}

Make sure to only use tables and columns from the schema above and write a
DuckDB SQL query
to answer the following question:
{{question}}

Please make sure to write a valid DuckDB SQL query.
```

A.2 Cost Analysis for Description Regeneration

Our database contains approximately 2,600 columns and 150 tables requiring description generation. Based on average input token count of 2,175 tokens per column description prompt and 625 tokens per table description prompt, the total estimated usage is approximately 5.65 million tokens. Using Gemini Flash pricing at $0.075 per million tokens, the total cost for complete description regeneration is approximately **$0.42**.