The Power of Constraints in Natural Language to SQL Translation

Tonghui Ren
Fudan University
TCHouse, Tencent Cloud
thren22@m.fudan.edu.cn

Chen Ke
Fudan University
cke23@m.fudan.edu.cn

Yuankai Fan
Fudan University
TeleAI, China Telecom
fanyuankai@fudan.edu.cn

Yinan Jing
Fudan University
jingyn@fudan.edu.cn

Zhenying He
Fudan University
zhenying@fudan.edu.cn

Kai Zhang
Fudan University
zhangk@fudan.edu.cn

X.Sean Wang
Fudan University
xywangCS@fudan.edu.cn

ABSTRACT

Current large language model (LLM)-based Natural Language to SQL (NL2SQL) approaches typically rely on the database schema and partial data values for the translation. These approaches are unable to use sufficient data for accurate database understanding due to limitations in data selection methods, and they cannot input the entire database due to the limited context window sizes of LLMs. This insufficient data integration may result in an incomplete understanding of the database, leading to semantically incorrect SQL generation. In this paper, we introduce REDSQL, a novel plug-and-play framework that refines the predicted SQL by utilizing the entire database in the refinement process. The core idea of REDSQL is to enhance SQL refinement by identifying potential errors based on the database content, which is achieved by applying constraints on the input relations of query operations. LLMs can refine the SQL using SQL-related information extracted by REDSQL, which provides concise and informative insights into the database. Additionally, REDSQL enhances schema semantics by integrating data profiling for more effective database utilization. Our experiments demonstrate that REDSQL consistently improves the performance of existing NL2SQL approaches across five benchmarks. Specifically, REDSQL elevates the accuracy of CODES to 67.3% (+8.8%) and PURPLE to 67.7% (+11.1%) on the BIRD benchmark.

PVLDB Reference Format:
Tonghui Ren, Chen Ke, Yuankai Fan, Yinan Jing, Zhenying He, Kai Zhang, and X.Sean Wang. The Power of Constraints in Natural Language to SQL Translation. PVLDB, 18(7): 2097 - 2111, 2025.
doi:10.14778/3734839.3734847

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/httdty/REDSQL_VLDB.

1 INTRODUCTION

SQL is a robust yet complex query language used for interacting with databases. Due to the complexity and non-intuitive nature of database design, it is challenging for non-professionals to write SQL. The Natural Language to SQL (NL2SQL) translation helps these users easily access database data.

Large language models (LLMs) have improved NL2SQL performance [53, 72]. Some works [52, 53] fine-tune LLMs on NL2SQL corpora for focused SQL translation and high accuracy. Other approaches [24, 36, 49, 72] use prompting strategies to instruct general-purpose LLMs to generate SQL without parameter updates.

However, the SQL queries generated by existing approaches may be incorrect **due to a lack of understanding of the database** without sufficiently considering its content. Existing LLM-based NL2SQL approaches often rely on natural language (NL) questions, database schema, and a small amount of retrieved database content for translation. Recent works attempt to improve database comprehension by integrating advanced database content retrieval techniques, which assist LLMs in NL2SQL translation [60]. However, these approaches typically depend on the string and embedding similarity [53, 59, 60, 83, 89] between NL questions and database content, which may overlook important database relationships or fail to retrieve essential information limited by the retrieval method. This may result in either an incomplete understanding of the database or an overload of irrelevant data, which hinders the accuracy of NL2SQL translation. Additionally, inputting the entire database into current LLMs is impractical due to limited context window sizes and high computational costs.

Figure 1 illustrates a "buggy" NL2SQL translation from the BIRD benchmark [55], caused primarily by insufficient data integration. Four specific errors are highlighted. Error ① arises from an incorrect predicate: the value "di Resta" is in the *surname* column instead of *driverRef* in DRIVERS. Consequently, error ② occurs when the nested query returns NULL, since "di Resta" is not found in *driverRef*. This leads to error ③, where an invalid equality comparison attempts to match an INTEGER with NULL, causing the entire query to yield NULL (error ④). It is challenging for LLMs to discern that "di Resta" is not in *driverRef* or notice the wrong equality comparison due to the limited content. Such errors can be constrained by integrating data accessibility into the refinement process, allowing for improved database understanding.

It is important but challenging for NL2SQL translation to incorporate sufficient database content. To address this issue, we propose to verify the predicted SQL based on the database content without providing all content to the LLMs. We notice that existing DBMSs use constraints on data write operations (insert, update, delete) to preserve business semantics [4, 6, 84], and the constraints violation report can help identify the wrong write operations. In this paper, we propose to use constraints on data for read operations (select),

Corresponding authors: Zhenying He, X.Sean Wang

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734847

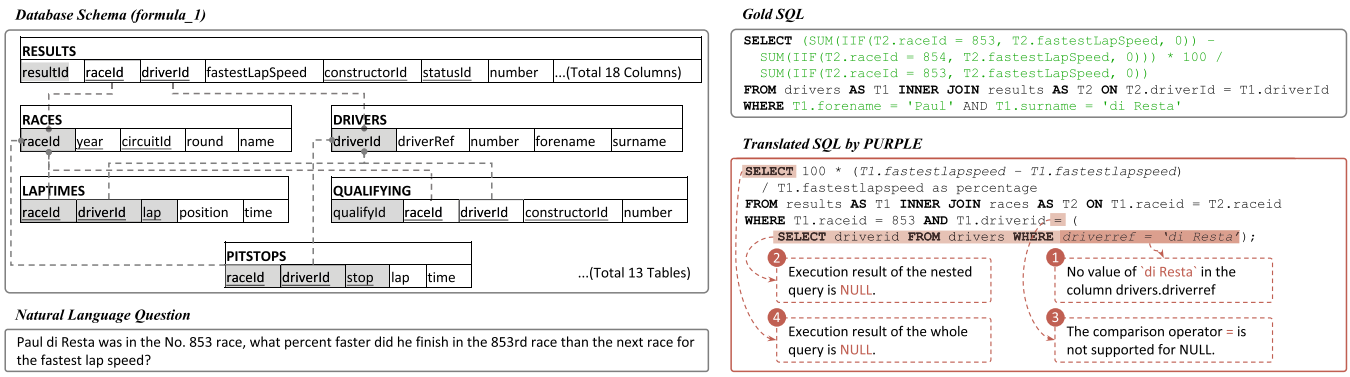

Figure 1: An example of NL2SQL translation task from Bird.

ensuring that operations align with business semantics. Such constraints can be evaluated based on the **entire database**, and the constraints violation report contains information from the database content to help refine the SQL.

Several studies have explored refinement techniques [18, 66, 83] for NL2SQL, but they primarily rely on an explanation generated by LLMs and make the LLMs refine the SQL based on the explanation. These studies concentrate on the SQL aspects rather than the data aspects. Additionally, these existing refinement strategies heavily depend on LLMs, which may limit their reliability and effectiveness in addressing data-related errors for accurate SQL generation.

There are two challenges in enabling LLMs to refine SQL based on the database content. The main challenge is: ***Where are the errors?*** Identifying potential errors in SQL based on database content is challenging. Firstly, most errors are not detectable by the DBMS (executable but semantically incorrect SQL). Additionally, LLMs cannot locate errors based on the limited database content, as discussed above. The other challenge is: ***How to fix the errors?*** LLMs need to address the errors within the SQL. However, some errors might be complex, requiring specific designs to assist LLMs in identifying and correcting errors with the help of the data.

In this paper, we aim to refine SQL generated by existing NL2SQL approaches, focusing on the data content of the database. To tackle the aforementioned challenges, we introduce REDSQL (**RE**fine by **D**ata for NL2**SQL**), a plug-and-play framework designed to detect potential semantic errors in SQL and prompt LLMs to generate a refined version. The core idea of REDSQL is that, while we cannot input all content into the LLMs, we can validate the predicted SQL based on the entire database. We design a **constraints verification framework** that is based on the alignment between query operations and relational calculus. This framework supports the customization of constraints for various query operations in relational databases, and such constraints can help maintain the business semantics of queries. We implement a general set of constraints applicable to diverse workloads, with the flexibility to add workload-specific constraints in real-world applications. To enable LLMs to understand potential semantic errors and refine SQL, REDSQL incorporates a pipeline to invoke LLMs for SQL refinement.

REDSQL can be integrated with any LLM-based NL2SQL approach in a plug-and-play style. When introducing a new database, REDSQL first builds enhanced column annotations. Once integrated into an existing NL2SQL approach, REDSQL takes the predicted SQL and validates it against the database content based on the defined constraints. It then generates a verification report and an informative context, which guides the LLM to refine the SQL.

The contributions of this work are as follows:

- We propose applying constraints to preserve the business semantics of queries for NL2SQL, helping LLMs refine predicted SQL based on the entire database.
- We implement a constraints verification framework for query operations, including a general set of constraints suitable for various workloads, with support for adding workload-specific constraints in real-world applications.
- We introduce REDSQL as a plug-and-play framework for SQL refinement. REDSQL identifies potential errors using constraints verification and provides SQL-related context to assist LLMs in refining SQL.
- Our experiments show that REDSQL consistently improves the performance of various NL2SQL approaches, achieving new state-of-the-art (SOTA) performance. Specifically, REDSQL enhances CODES to 67.3% (+8.8%) and PURPLE to 67.7% (+11.1%) translation accuracy on the Bird benchmark.

This paper is structured as follows: Section 2 introduces related work; Section 3 provides an overview of REDSQL; Section 4 details its components; Section 5 presents evaluation experiments; and Section 6 concludes with future research directions.

2 RELATED WORK

2.1 NL2SQL

NL2SQL tasks have been studied for decades. Early works [38, 45, 50, 51, 75, 78, 79, 91] focused on mapping NL terms to schemas, which limited NL understanding and generalization to cross-domain situations. With the rise of LLMs, most current NL2SQL approaches are based on LLMs. We categorize these as fine-tuning-based and prompting-based approaches.

Fine-tuning-based NL2SQL: Works such as [11, 14, 32, 33, 41, 44, 53, 54, 58, 69, 74, 76, 77, 89, 94], adapt pre-trained LLMs for NL2SQL by fine-tuning. Studies like [28–31, 52, 92] integrate ranking modules to boost accuracy. However, fine-tuning-based

methods are limited by model size and training corpus quality, causing performance drops in shifting workloads.

Prompting-based NL2SQL: Prompting-based approaches are divided into zero-shot and few-shot, depending on whether demonstrations are provided. Zero-shot approaches [17, 24, 42, 59] use hand-crafted instructions to improve translation accuracy. Few-shot approaches, such as [5, 36, 47, 49, 65, 67, 70, 72, 85, 88], include NL2SQL demonstrations to enhance SQL generation. However, these approaches often face limitations due to a small set of demonstrations, restricting their generalization across diverse workloads.

Existing LLM-based NL2SQL approaches often lack sufficient data for effective translation and struggle with processing entire databases. This limited data integration causes inaccuracies in SQL generation, as LLMs fail to understand database and SQL semantics.

2.2 SQL Semantic Debugging

SQL semantic debugging analyzes and corrects SQL semantics, going beyond syntax checks. It involves two types: one identifies discrepancies between a given SQL and a gold SQL, while the other detects potential semantic errors without the gold SQL. We also explore how LLMs are used to enhance SQL semantic debugging.

Semantic Debugging with Gold SQL: Identifying errors based on the gold SQL helps users correct their SQL. Existing works [16, 19, 37, 43, 61] propose methods for error identification and correction. However, in NL2SQL applications, access to a gold SQL for comparison is infeasible.

Semantic Debugging without Gold SQL: Detecting potential semantic errors is challenging without the gold SQL. Research [12, 87] aims to identify operations that no database model can satisfy, offering a safe yet sometimes impractical approach for real-world applications. Considering the actual database content helps detect more potential bugs. Such techniques are useful in educational settings and for categorizing common errors among SQL users [3, 13, 39, 68, 80–82]. These works identify error patterns through empirical analysis, focusing on user SQL rather than NL2SQL applications. Moreover, some studies [23, 40, 48] use visualization to explain the SQL execution process. These methods enhance user understanding of SQL semantics through visual representations of data operations. Although beneficial for users, integrating such visual insights into LLMs remains a challenge.

Semantic Debugging by LLMs: Some studies have employed LLMs for SQL semantic debugging, similar to REDSQL. Existing methods [15, 18, 66, 83, 88] often delegate the entire debugging process to LLMs. These methods involve using LLMs to interpret and then refine SQL based on the explanations. This direct approach can slightly enhance translation accuracy. However, identifying bugs without sufficient data access poses challenges for LLMs. Moreover, current approaches often depend on prior NL2SQL methods.

3 REDSQL OVERVIEW

Figure 2 provides an overview of REDSQL, including five main steps, as the **SQL Execution**, **Constraints Verification**, **Context Expansion**, **SQL refinement** and **Documentation**. The red rounded rectangle is the online refinement process, while the blue rounded rectangle is the documentation process.

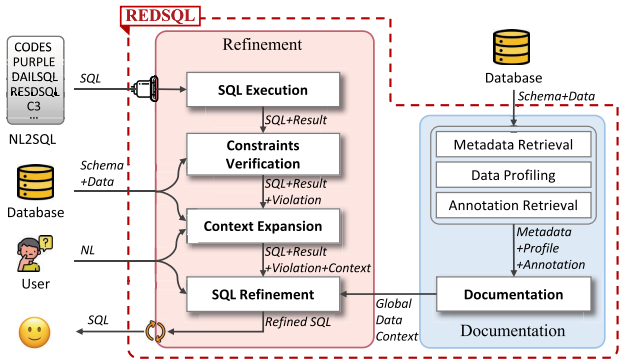

Figure 2: Overview of REDSQL.

Before introducing REDSQL, it is important to note that existing NL2SQL approaches mainly rely on either a *Schema Only* or a *Schema + Retrieved Data* strategy. These strategies fail to fully utilize the database content, leading to three main disadvantages:

- **Inability to Accurately Capture Schema Semantics.** For example, in Figure 1, a wrong predicate such as `driverref = 'di Resta'` occurs due to the incorrect interpretation of the column *driverref* without sufficient enrichment of the database content.
- **Failure to Properly Integrate Data Operations.** For example, an incorrect comparison between *driverid* (INTEGER) and NULL (executed by the subquery) arises because LLMs cannot recognize the execution result of the subquery without knowledge of the database content.
- **Incorrect Understanding of SQL Semantics.** For example, the generated SQL shown in Figure 1 is incorrect and goes undetected by LLMs because they cannot infer the execution process without access to the full database content.

In REDSQL, we address these issues by validating the predicted SQL against the entire database and then providing LLMs with detected potential errors and the relevant context for SQL refinement.

3.1 Online Refinement Process

The online refinement process of REDSQL consists of four steps: **SQL Execution**, **Constraints Verification**, **Context Expansion**, and **SQL Refinement**. Its inputs include the database, the NL question, and the initially predicted SQL. In addition, the **Documentation** step supplies enhanced column annotations that serve as the global data context for refinement. When REDSQL is integrated with existing NL2SQL approaches, the SQL can be refined repeatedly; each iteration uses the previously refined SQL as a new starting point. The final output is the refined SQL, which usually achieves higher accuracy than the initial prediction.

SQL Execution: The refinement process starts with the SQL execution to evaluate its executability. For a non-executable SQL, its execution results include exceptions reported by the DBMS. Conversely, for an executable SQL, its execution results become part of the SQL-related data. Collecting these responses from the DBMS helps LLMs verify the correctness of the SQL.

Constraints Verification: In this step, we detect potential semantic errors based on the data. Existing DBMSs primarily report syntax and execution errors but cannot identify semantic errors, which depend on the database data and the behavior of the corresponding DBMS. For instance, the query `ORDER BY` with `LIMIT 1` is commonly used to retrieve the minimum value. However, this operation may yield incorrect results if NULL values are present in the ranking column since some DBMSs rank NULL first in an `ORDER BY` operation. Although the SQL is executable, its semantics can vary depending on the database content (*e.g.,* whether NULL value in the `ORDER BY` column). The constraints verification validates the SQL by traversing the logical execution tree and evaluating it based on the database. Specifically, we implement constraints to the input relations for the operations. Detected constraint violations are classified into three levels: **INFO**, **WARNING**, and **ERROR**. Each violation report includes the violation level, location, and a detailed description.

Context Expansion: We notice that simply reporting detected potential errors is insufficient for effective SQL refinement by LLMs. Offering SQL-related information is beneficial as it provides informative context for refinement. During this step, we expand on the schema and values for the LLMs to facilitate the refinement step.

SQL Refinement: In the SQL refinement step, we develop a prompt in a zero-shot setting for invoking LLMs, preserving the generalization capability. The prompt includes instructions, the database schema, the predicted SQL, its execution results, the detected violations, additional expanded context, and enhanced column annotations. Note that the reported violations are hypothetical; it is up to the LLMs to decide whether to correct them. The number of refinement iterations is predefined, and once those iterations are exhausted, the refined SQL is returned as the final output.

3.2 Documentation Process

Before the SQL refinement tasks, REDSQL creates enhanced annotation for the database. The documentation step extracts three types of data as the source data and summarizes them as the documentation. The source data include the **Metadata** of the database, the **Data Profile** for the database, and the **Annotation** from the data dictionary. The summarized document serves as a global data context, representing the business semantics of the entire database.

Metadata Retrieval: The metadata includes details about tables, columns, and keys, providing a structural description of the schema. Most NL2SQL approaches leverage metadata for SQL translation.

Data Profiling: The data profile establishes a global view of the database by existing profile tools. Unlike typical NL2SQL methods that focus on extracting only matched entities from the NL question, our approach includes summarized statistical information to give LLMs a broader understanding of the database semantics.

Annotation Retrieval: Because table and column names can be ambiguous or uninformative, many NL2SQL tasks provide basic annotations. We collect these annotations to clarify the schema and support a foundational understanding.

Documentation: REDSQL synthesizes the metadata, data profile, and annotations into an enhanced column annotation. Directly utilizing raw data for the LLMs would be too verbose, costly, and lacking business semantics. Therefore, we generate a concise, semantically rich document through the LLM summarization process. This documentation serves as the global data context for subsequent NL2SQL refinement tasks.

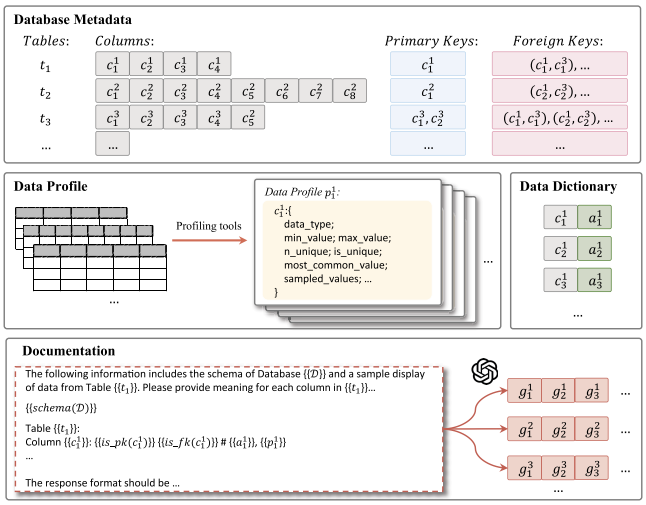

Figure 3: Documentation process of REDSQL.

4 METHODOLOGIES

In this section, we describe the detailed implementation of REDSQL. First, we describe the **Documentation** step, which produces a comprehensive database document used in the subsequent refinement. Then, we introduce the refinement process, outlining its four main steps: **SQL Execution**, **Constraints Verification**, **Context Expansion**, and **SQL Refinement**. REDSQL not only includes the errors raised by DBMS but focuses on identifying semantic errors in predicted SQL. By expanding SQL-related information, REDSQL offers LLMs a richer context for more accurate refinement.

4.1 Documentation

The objective of the documentation is to construct a global data context for SQL refinement. Expert SQL developers can write accurate queries because they understand SQL and the underlying database. Similarly, we aim to provide LLMs with comparable insights by creating a global data context for enhanced SQL refinement. Figure 3 illustrates the documentation pipeline.

4.1.1 Data Extraction. Given a database $\mathcal{D}$, we extract metadata about tables $\mathcal{T}$, columns $\mathcal{C}$, and keys $\mathcal{K}$. This metadata conveys structural information essential for representing the basic semantics of the database. We serialize the database schema as $schema(\mathcal{D})$ following the concise representation format in DIN-SQL [66].

To construct the global data context $\mathcal{G}$, we perform data profiling [1, 2, 62] on the entire database content $\mathcal{M}$. Data profiling captures statistical information about each column and reveals inter-column relationships or dependencies. In REDSQL, we use an existing data profiling tool[1] to generate a data profile $\mathcal{P}$. For each column c_i^j in table t_j, we obtain a profiling report p_i^j.

[1]https://github.com/ydataai/ydata-profiling *[Last accessed: 2025-04-13]*

Database column names typically use abbreviations, resulting in unclear semantics. Many databases include a data dictionary for clarification. Therefore, existing NL2SQL approaches often include a simple data dictionary to annotate the schema. We denote the annotation of c_i^j as a_i^j. Although these annotations are simple and do not convey the business semantics of the database, they play an important role in constructing the global data context.

For example, as illustrated in Figure 1, the database ***formula_1*** comprises 13 tables. The table DRIVERS has five columns: *driverId* (*Primary Key*), *driverRef*, *number*, *forename*, and *surname*. The annotation for the column *driverRef* is provided as "*driver reference name*". However, the annotation for *number* is simply "*number*", offering no additional semantic details.

4.1.2 Global Data Context Documentation. Data profile goes beyond raw statistics for data scientists, which can be utilized to infer potential business semantics. We aim for the LLMs to explicitly perform this task, ensuring that $\mathcal{G}$ contains the data-supported business semantics of the database. We notice that using metadata, data profiling, and column annotations to detail all available information contains duplicate content and lacks business semantics. Instead, we employ LLMs to generate a description for each column, serving as the global data context $\mathcal{G}$. This summary is more efficient for subsequent SQL refinement tasks since the summarization is performed offline and only needs to be executed once, making it reusable across multiple refinement tasks. Unlike traditional data profiling, which focuses on distribution and numerical summaries, our approach leverages LLMs to create concise descriptions that reveal clearer business semantics.

To generate $\mathcal{G}$, we construct a prompt for invoking LLMs as shown in Figure 3. The prompt contains four main parts: *instruction*, *schema*, *data*, and *formatting*. The *instruction* guides the LLMs to perform the column description task; the *schema* provides a basic structural overview of the database. The *data* part includes detailed metadata, profiling, and annotations for each column. The *formatting* guides the LLMs in structuring the response in a specific format, facilitating response parsing. To ensure the generated documentation sufficiently represents column semantics, we generate descriptions n times for each column and set an upper length bound τ_u to filter out long entries, which could otherwise consume excessive tokens in the online tasks.

For example, the global data context generated for the column *driverRef* is "*A reference name used internally for the driver.*", which concisely reflects its business semantics. For the numeric column *number* in DRIVERS (as shown in Figure 1), the summary is "*The racing number of the driver, if available*," which is not only rich in business semantics but also indicates that this value can be NULL.

4.2 SQL Execution

REDSQL is a framework that boosts the accuracy of existing LLM-based NL2SQL approaches without assuming specific implementation details or designs from prior NL2SQL approaches. Given an NL question $\mathcal{X}$ and a predicted SQL query $\mathcal{S}$ generated by any NL2SQL approach, REDSQL aims to refine $\mathcal{S}$ into $\mathcal{S}'$ such that the execution results of $\mathcal{S}'$ match those of the gold SQL $\mathcal{S}^*$.

Algorithm 1: Violation Collection Algorithm

```
Input  : Logical execution node n
Output : Detected violations V
Procedure CollectViolation(n, V):
    V ← []
    if IsLeaf(n) then
        // Leaf nodes do not contain violations
        return V
    else
        for each child ∈ IterChildren(n) do
            // Recursively collect violations
            MergeList(V, CollectViolation(child))
    // Check constraints for the current node
    MergeList(V, CheckConstraints(n))
    return V
```

For the predicted SQL $\mathcal{S}$, we first execute it on the target database. We record two types of responses: the *Exception* for non-executable SQL and the *Execution Results* for executable SQL, denoted as e.

DBMS raises exceptions for non-executable SQL (due to syntax errors, execution errors, etc.). We capture this response from the DBMS as errors in the non-executable SQL. We do not perform further constraints verification on the non-executable SQL since the SQL can not be executed for verification.

For executable queries, we collect the execution results. We retain only the first k tuples to limit the output size. If more than k tuples are retrieved, we log the first k and note how many are omitted. Although execution results do not convey the complete meaning of the SQL, this sample is valuable for assessing whether the response aligns with the NL question.

4.3 Constraints Verification

As previously mentioned, relying solely on the execution result e is insufficient for identifying all potential errors, especially those related to business semantics. Simply incorporating execution feedback yields only marginal improvements. A critical challenge lies in error detection, as LLMs often struggle to identify specific bugs caused by $\mathcal{M}$ without access to the entire database. Therefore, precise data knowledge is crucial for LLMs to detect and rectify potential errors in the SQL.

Traditional constraints for relational databases are typically designed for data write operations, ensuring the preservation of business semantics and data integrity [6, 8, 9, 25–27]. However, no constraints are applied to data read operations, as these queries do not modify the data, and it is assumed that SQL experts understand the business semantics of the queries. In NL2SQL systems, LLMs often lack full comprehension of the database semantics, leading to incorrect SQL generation. To address this, we propose to **integrate constraints into query operations**, assisting in finding potential errors based on database content. Unlike write operations, where constraints are evaluated post-operation to check for violations, read operations cannot rely on updated data. Instead, we propose applying constraints to the input relations of each operation to ensure that business semantics are preserved during query execution.

REDSQL provides a framework that supports all relational database operations by aligning them with relational calculus [20, 46,

Table 1: All constraints implemented in REDSQL. Function type: detects the data type of a value; Function is_null: checks whether a value is NULL; Function has_relation: identifies whether two columns have defined relationships; Function compatible: Determines whether the type of values is compatible with the operation being performed.

Operation	Type & Level	Formal Definition	Description
Predicate	**Predicate unsatisfiable** INFO	*Predicate node* n $\exists t \in R, (P(t))$;	P denotes the predicate in n, R denotes the input relation of n. Constraint: There exists a tuple t in R that satisfies P.
	Idle predicate ERROR	*Predicate node* n; $\exists A, B, (A \neq B)$;	A and B denote columns being compared within n. Constraint: A and B can not be the same columns.
CAST	**Data precision loss** WARNING	*CAST operation (to INTEGER) node* n; $\forall a \in A, (\text{type}(a) = \text{REAL} \rightarrow \lfloor a \rfloor = a)$;	A denotes columns involved in the `CAST` operation of n. Constraint: All values a in A, if of type REAL, must equal their floor value.
`JOIN` predicate	**`JOIN` abnormal** WARNING	*`JOIN` predicate node* n; $\text{has\_relationship}(A, B)$;	A and B denote the columns used in the JOIN predicate of n. Constraint: There must be a defined relational key constraint between A and B.
	Wrong `JOIN` ERROR	*`JOIN` predicate node* n; $\exists a \in A, \exists b \in B, (P(a, b))$;	A and B denote the columns used in the `JOIN` predicate P of n. Constraint: There must be overlapping values between A and B.
Query	**Output abnormal** WARNING	*Query node* n; $(R \neq \emptyset) \wedge (\exists t \in R, (\neg is\_null(t))) \wedge$ $(\forall t1, t2 \in R, (t1 \neq t2))$;	R denotes the input relation of n. Constraint: The relation R must not be empty, contain no NULL tuples, and have no duplicate tuples.
Calculation	**Wrong calculation** ERROR	*Calculation node* n; $\forall A, B, (\text{compatible}(\text{type}(A), \text{type}(B)))$;	A and B denote columns or values involved in calculations within n. Constraint: Operations between A and B must involve compatible data types.
Comparison	**Wrong comparison** ERROR	*Comparison node* n; $\forall A, B, (\text{compatible}(\text{type}(A), \text{type}(B)))$;	A and B denote the columns being compared within n. Constraint: The data types of A and B must be compatible.
Logical	**Predicates inconsistent** ERROR	*Logical node* n; $\exists t \in R, (P_s(t))$;	P_s represents predicates composed within n; R denotes the input relation of n. Constraint: There must exist a tuple t in R that satisfies P_s.
`SELECT`	**Uncertain projection** ERROR	*`SELECT` (with `GROUP BY`) node* n; $\forall t1, t2 \in R, (t1.A = t2.A \rightarrow t1.B = t2.B)$;	A denotes the `GROUP BY` column of n, B denotes columns in the `SELECT` clause of n; R denotes the input relation of n. Constraint: For all tuples have the same A value, they have the same B value.
`GROUP BY`	**Idle `GROUP BY`** ERROR	*`GROUP BY` node* n; $\exists t1, t2 \in R, (t1.A = t2.A \wedge t1 \neq t2)$;	A denotes the `GROUP BY` column of n; R denotes the input relation. Constraint: There must exist at least two tuples that have the same value for A.
`HAVING`	**Uncertain `HAVING`** ERROR	*`HAVING` predicate node* n; $\forall t1, t2 \in R, (t1.A = t2.A \rightarrow t1.B = t2.B)$;	A denotes the `GROUP BY` column of n, B denotes columns in the `HAVING` predicate of n; R denotes the input relation of n. Constraint: For all tuples have the same A value, they have the same B value.
`ORDER BY`	**Ranking abnormal** WARNING	*`ORDER BY` node* n; $\forall a \in A, (\neg \text{is\_null}(a))$;	A denote the `ORDER BY` column of n. Constraint: All values a in A must be non-null for valid ranking.

63, 64, 73, 86]. Leveraging this alignment, we model the verification process as an evaluation of the input relations for each operation. Specifically, constraints can be defined using logical forms for particular operations. These logical constraints are then applied to the corresponding input relations. Additionally, the completeness of REDSQL for all relational operations is guaranteed through the coverage of relational calculus across SQL operations.

Each operation n in the SQL execution process receives relations as inputs. The potential errors can be noticed if the added constraints are violated. It is important to notice that the constraints are applied in the execution process, and the constraints do not really stop the execution but only report the violation.

Algorithm 1 details the process for identifying all constraint violations in $\mathcal{S}$. The algorithm returns an empty list for the leaf nodes (tables, columns, aliases, and literal values) in the execution tree (lines 3-4), because these nodes do not represent operations requiring validation. For non-leaf nodes, the algorithm recursively collects violations by iterating through each child node of n (lines 5-7), merging detected violations into the violations list $\mathcal{V}$. Subsequently, violations of n are identified based on the input relations. To collect all violations in $\mathcal{S}$, REDSQL invokes the CollectViolation function with the root node of the execution tree.

We list all constraints that we implemented in Table 1. These constraints primarily focus on discrepancies related to data types, values, and uncertainties introduced by the DBMS, which is generally applicable to databases in real-world applications. In formulating these constraints, we reference studies on semantic debugging without a gold standard SQL [3, 13, 39, 68, 80–82] and incorporate common error patterns from LLM-generated SQL as noted in previous research [49, 66, 93]. These two aspects allow us to address errors typically introduced by humans and LLMs, enhancing the robustness of our approach. Furthermore, REDSQL is designed as a flexible framework capable of designing more constraints if only the operations are aligned with relational calculus. We only implement the constraints that are generally applicable in this paper. For example, the constraints **Uncertain projection**, **Idle `GROUP BY`**, and **Uncertain `HAVING`** address common `GROUP BY` issues. Specifically, **Uncertain projection** prevents including non-aggregated columns in a `SELECT` clause that uses `GROUP BY`. **Idle `GROUP BY`** removes unnecessary `GROUP BY` clauses on unique columns, and **Uncertain `HAVING`** ensures predicate columns in a `HAVING` clause are either aggregated or deterministic.

In practice, designing specific constraints tailored to a particular database or business requirement can further improve accuracy. For example, consider the data type of the column *time* in the table LAPTIMES, which is stored as TEXT (as shown in Figure 1). Sorting this column alphabetically can lead to incorrect results (*e.g.,* "52:18.842" is smaller than "5:03.706"). A custom constraint could prohibit `ORDER BY` on *time* while it remains of type TEXT, thereby maintaining business semantics. However, to preserve a cross-database approach and avoid overfitting, we exclude such workload-specific constraints, focusing instead on more general constraints applicable to various workloads.

We classify all violations into three levels: **INFO**, **WARNING**, and **ERROR**. This classification aligns with common practices in code compilers, which prioritize bugs based on their impact. The criteria for categorizing violations are based on the potential of the relational calculus to cause actual semantic errors.

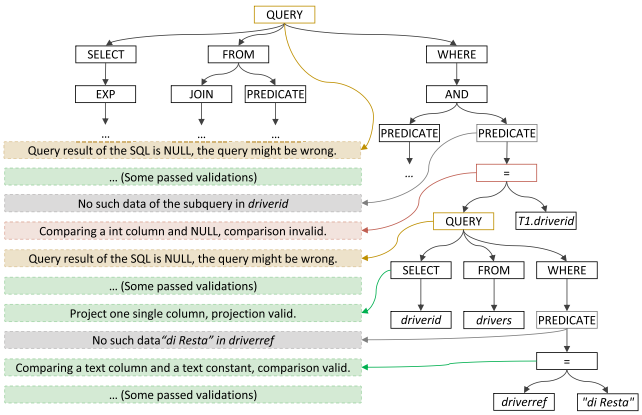

Figure 4: Constraints verification example: Green boxes show no violations; gray, yellow, and red boxes represent INFO, WARNING, and ERROR level violations, respectively

- **INFO:** This level is used to collect information about which operations cannot be satisfied by the input relations. While not indicating an abnormal or incorrect operation, showing instances at this level is crucial for identifying potential sources of higher-level errors.
- **WARNING:** At this level, we categorize operations that deviate from normal expectations based on the input. The operations identified at this level are uncommon in typical SQL workloads and might indicate potential errors.
- **ERROR:** This level includes operations that may not be directly executable but are implicitly handled in a default manner by the DBMS design. For example, some DBMSs permit dynamic data type operations (*e.g.,* SQLite), such as comparing a TEXT column with an INTEGER column, which can lead to unpredictable results. This level also includes inherently meaningless operations.

We demonstrate the constraints verification process for the SQL shown in Figure 1 as an example. Due to space limitations, not all steps are displayed. Five potential errors were identified in $\mathcal{S}$: The first violation is in the predicate `driverref = 'di Resta'`, where there is an **INFO**-level violation because the value "di Resta" does not exist in the column *driverRef*. The second detected violation is **WARNING**-level, as the result of the nested SQL is NULL. All other detected violations are listed in Figure 4.

Discussion: We highlight three key aspects of the violation verification process. First, although REDSQL flags potential errors through constraint violations, these issues do not necessarily indicate mistakes. For example, an empty (NULL) result might be correct, and the violation prompts LLMs to confirm whether the SQL matches user intent. Second, while constraint evaluation on input relations can be time-consuming, it typically only takes a few seconds for large databases in Bird and around ten seconds for the extremely large ScienceBenchmark [95]. Finally, REDSQL can integrate workload-specific constraints aligned with particular business and database requirements, offering a broad design space to accommodate unique database semantics.

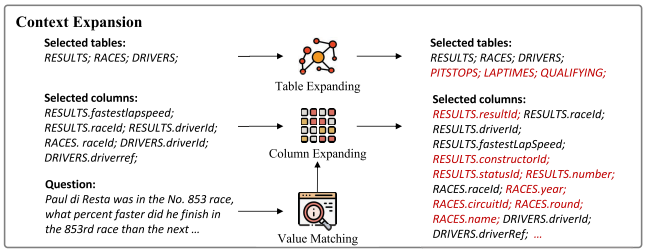

Figure 5: Context Expansion in REDSQL.

4.4 Context Expansion

We recognize that reporting detected violations is crucial for SQL refinement. However, LLMs sometimes struggle to correct the SQL adequately, especially when the predicted SQL is largely incorrect. For example, if the generated SQL omits certain tables required by the gold SQL, it is difficult for REDSQL to fix this omission based solely on violation information. To address this issue, REDSQL expands the context for SQL refinement by incorporating additional SQL-related information, including schema and values, providing a richer basis for SQL refinement.

4.4.1 Schema Expansion. Providing high schema recall helps LLMs choose the correct tables and columns during refinement. However, including comprehensive information about all tables and columns in the prompt can increase the cost of invoking LLMs and complicate schema linking [57, 70, 72, 93]. Unlike existing NL2SQL approaches, REDSQL aims to refine the $\mathcal{S}$ to $\mathcal{S}'$ rather than generating the SQL from scratch. On the one hand, we should emphasize the information about the tables and columns referenced in $\mathcal{S}$ to assist LLMs in checking if $\mathcal{S}$ accurately reflects the intended schema. On the other hand, these references also provide clues about the gold tables and columns in $\mathcal{S}^*$ [57, 70, 93]. REDSQL does not solely rely on $\mathcal{S}$ for table and column selection; instead, it expands the selection to ensure robust schema linking as shown in Figure 5.

Algorithm 2 shows the schema expansion process utilized in REDSQL. We do not include semantic techniques in this process, as we assume that $\mathcal{S}$ inherently contains semantic information [57]. This algorithm focuses solely on expanding the initially selected tables $\mathcal{T}^p$ to $\mathcal{T}^e$ and the selected columns C^p to C^e.

For table expansion, we rank tables not initially in $\mathcal{T}^p$ but related to those in $\mathcal{T}^p$, based on the count of adjacent tables (lines 4-5). We add all tables in $\mathcal{T}^p$ to $\mathcal{T}^e$ and then include the top $k_1 - |\mathcal{T}^p|$ related tables based on neighbor counts.

For column expansion, we focus on the columns belonging to tables in $\mathcal{T}^e$. For each t_i in $\mathcal{T}^e$, we perform a ValueMatch to identify matched values between the $\mathcal{X}$ and each column c^i. This matching uses a coarse-to-fine string matching algorithm based on BM25 and sentence edit distance [53]. We add all columns in C^p into C^e and then add the top $k_2 - |C^p|$ columns with the highest priority (lines 7-10). The priority calculation is shown in Table 2.

4.4.2 Value Expansion. Moreover, we utilize the extracted values in $\mathcal{L}$ of Algorithm 2. We incorporate potential gold column hints by including extracted values from $\mathcal{L}$. Additionally, we enrich $\mathcal{L}$ by expanding maximum and minimum values, as well as NULL, to

Algorithm 2: Schema Expansion Algorithm

Input : Initial selected tables $\mathcal{T}^p$; Initial selected columns $\mathcal{C}^p$; Database $\mathcal{D}$; NL question $\mathcal{X}$

Output: Expanded tables $\mathcal{T}^e$; Expanded columns $\mathcal{C}^e$; Local data context $\mathcal{L}$

1 **Procedure** ExpandSchema($\mathcal{T}^p$, $\mathcal{C}^p$, $\mathcal{D}$, $\mathcal{X}$):
2 $\mathcal{T}^e \leftarrow \mathcal{T}^p$; $\mathcal{C}^e \leftarrow \mathcal{C}^p$; $\mathcal{L} \leftarrow \{\}$;
// Expanding tables
3 **for each** $t_i \in \mathcal{T}^p$ **do**
4 **for each** $adj \in$ GetNeighbor(t_i) **do**
5 AddCount($\mathcal{D}$, adj)
6 MergeList($\mathcal{T}^e$, GetTopTables($\mathcal{D}$))
// Expanding columns
7 **for each** $t_i \in \mathcal{T}^e$ **do**
8 **for each** $c_j^i \in t_i$ **do**
9 $\mathcal{L}[c_j^i] \leftarrow$ ValueMatch($\mathcal{X}$, c_j^i)
10 MergeList($\mathcal{C}^e$, GetTopColumns($\mathcal{L}$, $\mathcal{C}^i$))
// Expanding local data context
11 $\mathcal{L} \leftarrow$ ExpandData($\mathcal{L}$)
12 **return** $\mathcal{T}^e$, $\mathcal{C}^e$, $\mathcal{L}$

Table 2: Column Expansion Priority.

Priority	Description	Example
Highest	Columns in $\mathcal{C}^p$	DRIVER.*driverRef* (used in $\mathcal{S}$)
High	Columns in $\mathcal{K}$	DRIVER.*driverid* (primary key)
Medium	Columns in $\mathcal{L}$	DRIVER.*surname* (value matched)
Low	Other columns in $\mathcal{C}$	DRIVER.*number*

provide a more comprehensive data context related to the columns in $\mathcal{C}^e$ (line 11). The function ExpandData enhances the local data context by highlighting insightful data within the column. This approach helps the SQL refinement by providing a richer context for the LLMs during the refinement process.

4.5 SQL Refinement

The final step is refining SQL $\mathcal{S}$ to $\mathcal{S}'$ by invoking LLMs. We design this step to guide the LLMs in refining $\mathcal{S}$ through a structured prompt. Additionally, we incorporate the database adaptation module [72] to process the responses of LLMs.

We design a prompt to facilitate SQL refinement, which is utilized when invoking LLMs. As illustrated in Figure 6, the prompt consists of four main components: **Instruction** is consistent for all refinement tasks and aimed at directing the LLMs regarding the required actions. **Overall Schema** aligns with DIN-SQL [66] and offers a concise representation of the schema to provide structural information about the database. **Refinement Task** presents specific details about the current SQL refinement task, including the NL question $\mathcal{X}$ and the predicted SQL $\mathcal{S}$. This provides the LLMs with the current task information. **RED Prompt** comprises the selected tables $\mathcal{T}^e$, selected columns $\mathcal{C}^e$, the corresponding global data context $\mathcal{G}$, the local data context $\mathcal{L}$, and the violations $\mathcal{V}$. **RED Prompt** is the core of the prompt in REDSQL, as it helps LLMs in identifying real semantic errors based on $\mathcal{M}$.

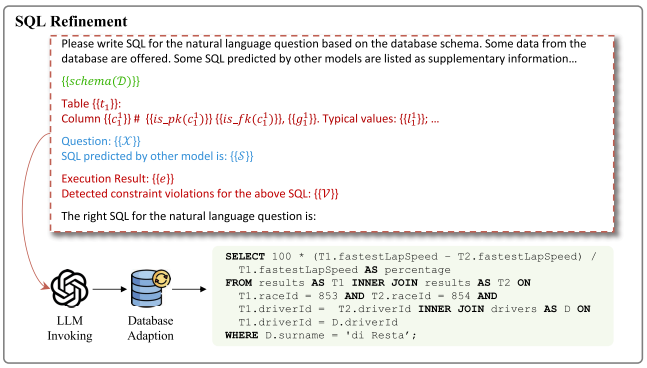

Figure 6: SQL Refinement in REDSQL.

There are two refinement strategies in REDSQL: *all refinement* and *bug-only refinement*. *All refinement* involves refining all received SQL, while *bug-only refinement* focuses on refining only the SQL containing potential errors, including execution errors and detected potential semantic errors. These two refinement strategies can be combined for multi-turn refinements.

We employ a zero-shot prompting strategy to maintain generalization. While advanced methods such as few-shot prompting could potentially enhance performance, the flexibility of constraint violations in our approach makes it challenging to create representative demonstrations. Moreover, few-shot methods typically increase the LLM usage costs. Therefore, REDSQL exclusively utilizes zero-shot prompting. Exploring advanced few-shot prompting strategies for REDSQL remains an interesting direction for future research.

5 EXPERIMENTS

In this section, we evaluate REDSQL by integrating it with various LLM-based NL2SQL approaches. We test the robustness of REDSQL under different hyperparameters, explore its performance with multiple foundation LLMs, and perform ablation studies to assess the contributions of each REDSQL component.

5.1 Experimental Setup

5.1.1 Benchmarks. We primarily evaluate REDSQL on the challenging Bird benchmark [55]. Most of our experiments use the Bird development set. We also include the simpler Spider benchmark [90] and its three variants, Spider-DK (DK) [35], Spider-Realistic (Realistic) [22], and Spider-SYN (SYN) [34], to explore domain adaptation and linguistic robustness. Additionally, we evaluate REDSQL on complex ScienceBenchmark (Science) [95], converting the original database to SQLite for consistency with our baselines.

Bird: This benchmark contains 12,751 NL2SQL pairs across 95 large-scale databases, highlighting challenges such as noisy database values and the need for external knowledge. It emphasizes the importance of understanding database values to generate accurate queries, aligning with the motivations of REDSQL.

Spider: This benchmark includes 10,181 questions and 5,693 unique SQL queries across 200 multi-table, cross-domain databases. Spider focuses on evaluating the multi-table database schema understanding of NL2SQL approaches.

Spider-DK, Realistic, and SYN: These three Spider variants investigate different aspects of domain adaptation and linguistic variation. DK requires domain-specific knowledge, Realistic alters text-schema alignment by removing direct schema references, and SYN rephrases questions to evaluate paraphrasing robustness.

Science: This benchmark involves domain-specific knowledge for real-world application queries based on large databases. We include it to evaluate REDSQL in complex scenarios further. Since all baseline methods are implemented in SQLite, we convert the original database to SQLite for evaluations.

5.1.2 Evaluation Metrics. For all benchmark evaluations, we primarily focus on the EXecution-match accuracy (EX), and we report the Valid Efficiency Score (VES) for Bird.

EX [55, 90] is the official evaluation metric used across all mentioned benchmarks, designed to assess whether the predicted SQL yields results that align with those of the gold SQL.

VES [55] measures SQL efficiency by comparing the execution time of predicted queries to that of the gold SQL. We perform 100 execution trials for VES on Bird. Although REDSQL mainly refines SQL for accuracy, we also investigate whether constraints can encourage more efficient SQL by leveraging database content.

5.1.3 NL2SQL Approaches. We integrate REDSQL with six LLM-based NL2SQL approaches, including both prompting-based and fine-tuning-based strategies. We want to evaluate whether REDSQL can consistently enhance the performance of various NL2SQL methods regardless of their underlying approach. Below are the selected existing NL2SQL approaches:

RESDSQL [52] employs a fine-tuning strategy for NL2SQL tasks. RESDSQL introduces a schema ranker and skeleton-aware decoding to improve the performance of T5 [71] on NL2SQL.

CODES [53] is designed to fine-tune StarCoder [56] using a collection of SQL data and domain-specific augmented NL2SQL data to improve the capabilities of StarCoder in NL2SQL tasks. We utilize the SFT-15B version of CODES for our evaluation.

C3 [24] is a zero-shot prompting approach using ChatGPT, which employs clear prompts, calibration with hints, and consistency in output to enhance the performance of LLMs on NL2SQL tasks.

DAILSQL [36] adopts a few-shot prompting strategy. It improves LLM performance by extracting demonstrations based on the question and SQL keywords. We use the self-consistency version of DAILSQL, which has shown superior performance.

PURPLE [72] also follows a few-shot prompting strategy. PURPLE enhances the SQL writing capabilities of LLMs by extracting demonstrations based on the composition of SQL logical operators. We implement PURPLE based on the training set of Bird and employ Longformer [10] for the larger context window.

SUPERSQL [49] is a hybrid approach that selects modules from existing NL2SQL techniques to maximize performance. Although SUPERSQL incorporates a refinement stage, we evaluate whether REDSQL can further boost its performance.

5.1.4 Other Refinement Frameworks. Several works have proposed SQL refinement for NL2SQL tasks [15, 66, 83, 88]. These approaches incorporate the refinement step into their pipeline. For comparison, we extract and implement the refinement strategies from DIN-SQL [66], MAC-SQL [88], and CHESS [83].

DIN-SQL [66]: The refinement process in DIN-SQL involves prompting LLMs to review the generated SQL. DIN-SQL uses a few-shot prompt and applies refinement to all predicted SQL. The refiner of DIN-SQL is referred to as DIN-Ref.

MAC-SQL [88]: MAC-SQL refines SQL only if it is invalid or produces empty (or NULL) results. It utilizes a zero-shot prompting strategy for the refinement, and its refiner is referred to as MAC-Ref.

CHESS [83]: CHESS refines SQL containing syntactic errors or yielding empty results. It employs a few-shot prompting strategy for SQL refinement. We extract its SQL refinement tool for comparison, referred to as CHESS-Ref.

5.1.5 Implementation Details. We primarily implement REDSQL, DIN-Ref, MAC-Ref and CHESS-Ref using ChatGPT (gpt-3.5-turbo)[2] by default. Additionally, we evaluate the performance of REDSQL when based on other LLMs, including DeepSeek (DeepSeek-V2) [21], DeepSeekCoder (DeepSeek-Coder-V2) [96], LLAMA3 (LLAMA3-70B-Instruct)[3], and QWen-72B-Instruct (QWen) [7], which reflect the performance of REDSQL on privately deployed LLMs. We also evaluate based on GPT4 (gpt-4o)[4] for higher performance evaluation. Our experiments are conducted on an Ubuntu 18.04 system with a 64-core CPU, 512GB of memory, and 8 NVIDIA A100 GPUs.

We typically conduct two rounds of refinement on the predictions for Bird. The first round applies *all refinement* strategy, and the second round applies *bug-only refinement* strategy. For the Spider benchmark, we refine only the buggy SQL given the lower complexity. We set the hyper-parameters for schema expansion at $k_1 = 6$ and $k_2 = 40$. Section 5.2 presents a detailed discussion on hyper-parameter settings.

5.2 Overall Performance

We evaluate the effectiveness of REDSQL by integrating it with existing NL2SQL approaches across various benchmarks. This evaluation allows us to measure the generalization and enhancements REDSQL brings to different NL2SQL approaches.

We evaluate REDSQL by integrating it with 6 NL2SQL approaches and reporting the EX and VES on the Bird benchmark, as shown in Table 3. Across all tested NL2SQL approaches, REDSQL consistently improved translation accuracy on Bird, with gains exceeding 5% in all cases. Notably, some NL2SQL approaches that initially struggled with the Bird benchmark achieved accuracies above 59%, surpassing the highest existing baseline of 58.5%. Specifically, REDSQL increased the EX for RESDSQL by 18.3%, demonstrating improvements even when paired with NL2SQL approaches not designed for Bird. Additionally, with REDSQL, CODES reached an EX of 66.2%. REDSQL enhances existing NL2SQL approaches by improving database data understanding, helping LLMs identify semantic errors in SQL.

DIN-Ref cannot consistently improve the base NL2SQL approach performance, as the refinement strategy relied on LLMs for error detection without providing additional data-related information. This limitation often leaves LLMs without the necessary insights to identify potential errors effectively, leading to incorrect SQL refinement. Conversely, MAC-Ref and CHESS-Ref only refine SQL

[2]https://openai.com/chatgpt/overview/ *[Last accessed: 2025-04-13]*
[3]https://ai.meta.com/blog/meta-llama-3/ *[Last accessed: 2025-04-13]*
[4]https://openai.com/index/gpt-4/ *[Last accessed: 2025-04-13]*

Table 3: Accuracy & Efficiency on Bird Dev.

Refiner	Metrics	RESDSQL [52]	CODES [53]	C3 [24]	DAILSQL [36]	PURPLE [72]	SUPERSQL [49]
Baseline	EX	43.9	58.5	50.2	55.9	56.6	58.5
	VES	44.2	60.3	50.9	57.2	58.1	60.4
DIN-Ref [66]	EX	47.7 (+3.8)	57.7 (-0.8)	49.6 (-0.6)	54.5 (-1.4)	55.5 (-1.1)	57.8 (-0.7)
	VES	47.6 (+3.4)	60.8 (+0.5)	50.3 (-0.6)	55.7 (-1.5)	57.5 (-0.6)	59.7 (-0.7)
MAC-Ref [88]	EX	47.0 (+3.1)	59.4 (+0.9)	51.1 (+0.9)	56.8 (+0.9)	57.0 (+0.4)	59.8 (+1.3)
	VES	47.1 (+2.9)	61.1 (+0.8)	52.0 (+1.1)	58.2 (+1.0)	58.5 (+0.4)	61.6 (+1.2)
CHESS-Ref [83]	EX	49.3 (+5.4)	59.6 (+1.1)	52.1 (+1.9)	57.3 (+1.4)	57.5 (+0.9)	60.0 (+1.5)
	VES	49.3 (+5.1)	61.5 (+1.2)	52.5 (+1.6)	58.8 (+1.6)	59.7 (+1.6)	61.8 (+1.4)
REDSQL (ours)	EX	**62.2 (+18.3)**	**66.2 (+7.7)**	**59.4 (+9.2)**	**61.7 (+5.8)**	**64.0 (+7.4)**	**63.6 (+5.1)**
	VES	**63.3 (+19.1)**	**70.4 (+10.1)**	**61.3 (+10.4)**	**64.6 (+7.4)**	**66.3 (+8.2)**	**67.1 (+6.7)**

Table 4: Accuracy on Spider Dev.

NL2SQL	Baseline	DIN-Ref	MAC-Ref	CHESS-Ref	REDSQL (ours)
RESDSQL	84.1	83.6 (-0.5)	83.9 (-0.2)	84.2 (+0.1)	**84.8 (+0.7)**
CODES	84.9	83.8 (-1.1)	85.3 (+0.4)	85.5 (+0.6)	**85.5 (+0.6)**
C3	82.0	80.0 (-2.0)	82.2 (+0.2)	82.2 (+0.2)	**82.8 (+0.8)**
DAILSQL	83.6	81.3 (-2.3)	83.5 (-0.1)	83.7 (+0.1)	**83.8 (+0.2)**
PURPLE	**84.8**	83.2 (-1.6)	84.2 (-0.6)	84.5 (-0.3)	**84.8 (+0.0)**
SUPERSQL	87.0	84.8 (-2.2)	86.9 (-0.1)	**87.1 (+0.1)**	**87.1 (+0.1)**

queries with execution errors or empty results to avoid an obvious negative impact. The existing SQL refinement methods lack a deeper understanding of the target database. The absence of detailed data-related information in these methods limits their performance.

We observe that **REDSQL performs optimally when integrated with NL2SQL approaches that utilize different LLMs.** Specifically, CODES and SUPERSQL initially exhibit similar performance levels but show different improvements when augmented with REDSQL. CODES is fine-tuned based on StarCoder [56] and shows a larger improvement compared to SUPERSQL, which is based on GPT. We attribute this difference to the refinement process acting as a model ensemble technique. Previous research [83, 93] has explored the benefits of collaborating with different LLMs for improved NL2SQL performance. REDSQL is a potential way to achieve the ensemble strategy. For an in-depth analysis of how different foundational models affect performance, we conduct more detailed model selection experiments in Section 5.4.

In addition to improving EX, REDSQL also enhances VES for existing NL2SQL approaches. By incorporating data-related information, LLMs generate more precise SQL that often runs more efficiently. The larger improvement in VES compared to EX suggests that REDSQL helps generate more efficient SQL, boosting performance in both accuracy and query execution efficiency.

We also evaluate REDSQL on Spider, as shown in Table 4. Given the lower complexity of Spider, the improvements brought by REDSQL are not as obvious as those observed with Bird. This suggests that the additional data-related information provided by REDSQL contributes less in simpler scenarios (where database semantics can be captured by schema information alone). However, REDSQL consistently outperforms other refinement methods and achieves the highest translation accuracy.

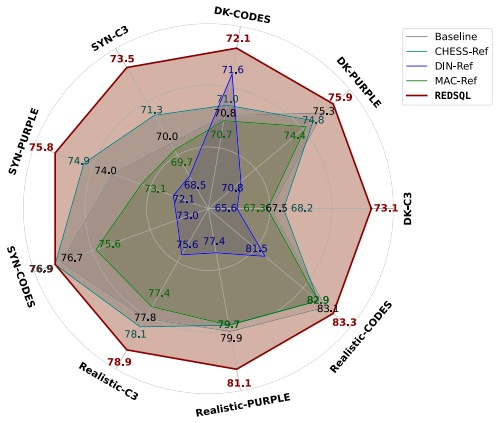

Figure 7: Accuracy on DK, SYN and Realistic.

To further evaluate whether REDSQL can manage complex scenarios based on the Spider, we extend our evaluation to include its variants: DK, SYN, and Realistic. These variants introduce additional complexities in refining SQL under challenging conditions.

We evaluated three distinct NL2SQL approaches: CODES (a fine-tuning approach), C3 (a zero-shot prompting approach), and PURPLE (a few-shot prompting approach), on the three variations of Spider. Figure 7 shows their EX results in both baseline and refined forms, comparing three refinement techniques for each approach and variant. We find that REDSQL consistently helps all evaluated baseline NL2SQL approaches achieve the highest scores across all variations of Spider. This success is attributed to the challenge in data understanding introduced by modifications to the NL questions in these variations. In contrast, other refinement methods suffer performance drops because they provide insufficient context for LLMs to accurately locate SQL errors. Interestingly, DIN-Ref performs well with CODES on the DK variation. We think this is because DK focuses more on domain knowledge, and the GPT model can enhance CODES in domain-specific contexts. MAC-Ref and CHESS-Ref do not achieve such high accuracy because they only refine non-executable SQL and SQL with empty results, limiting the improvement the model ensemble brings.

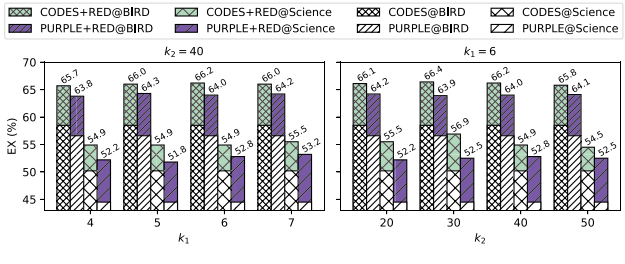

Figure 8: Accuracy of REDSQL with various k_1 and k_2.

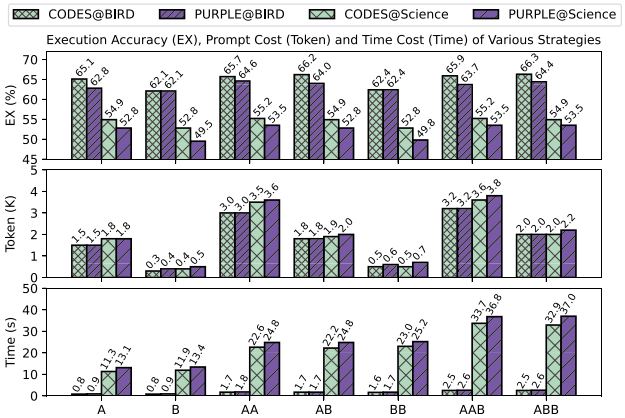

Figure 9: Accuracy and Cost Trade-Off of REDSQL under Different Refinement Strategies.

5.3 Performance and Cost Analysis

We further investigate how different hyper-parameters affect REDSQL. Specifically, we investigate the schema expansion algorithm that retains the top-k_1 tables and top-k_2 columns, as well as the optimal number of refinement iterations. We also evaluate performance–cost trade-offs by measuring prompt preparation time and prompt length under different refinement strategies. Furthermore, we introduce Science in our cost measurement experiments to assess the performance and cost of REDSQL in large-scale databases. These experiments are carried out using CODES and PURPLE, respectively representing the best fine-tuning-based and prompting-based approaches when integrated with REDSQL.

Figure 8 shows how performance varies with different k_1 and k_2 settings on both Bird and Science. The left side of the figure fixes $k_2 = 40$ and varies k_1, while the right side fixes $k_1 = 6$ and varies k_2. In all scenarios, REDSQL consistently improves the baseline NL2SQL models, indicating that it is robust to hyper-parameter choices. We attribute this robustness to the strong capacity of the baseline models to include the correct tables and columns in the predicted SQL ($\mathcal{S}$). On the more complex Science benchmark, REDSQL improves EX of CODES by more than 4.3% (from 50.2%) and EX of PURPLE by more than 7.3% (from 44.5%). In our experiments, we set $k_1 = 6$ and $k_2 = 40$ by default, effectively balancing high schema recall with limited token cost. Notably, for strong NL2SQL approaches such as CODES, a relatively small k_2 works well because of their high-precision SQL generation. However, a larger k_1 is beneficial, as existing NL2SQL approaches sometimes omit necessary tables in their generated SQL.

Within REDSQL, we implement two refinement strategies: *all refinement* (A) and *bug-only refinement* (B). In our experiments, we apply the A strategy first to propagate the global data context across all queries. The B strategy then follows, focusing on queries flagged by the constraints verification step.

As shown in Figure 9, higher token consumption generally leads to improved accuracy. However, CODES and PURPLE each achieve their highest accuracy under different multi-turn sequences. Specifically, ABB for CODES and AA for PURPLE on Bird, AA and AAB for CODES and AA, AAB, and ABB for PURPLE on Science. To balance performance with efficiency and prevent overfitting to any specific NL2SQL approach, we choose the AB refinement strategy as the default for REDSQL. We also analyze the prompt lengths of other refinement strategies for comparison. DIN-Ref exceeds 4,000 tokens, MAC-Ref exceeds 2,000 tokens, and CHESS-Ref exceeds 7,000 tokens. Notably, MAC-Ref and CHESS-Ref require only a few hundred tokens for refinement on average, as they focus solely on syntactic errors or empty-result SQL. In contrast, REDSQL provides a more economical and effective approach to SQL refinement, achieving higher accuracy with balanced token consumption.

We also evaluate the prompt generation time for REDSQL, as shown at the bottom of Figure 9. On Bird, REDSQL generates prompts in approximately one second per refinement on average. For the larger databases in Science, the average prompt generation time increases to around ten seconds per refinement. Accelerating the constraints verification process within REDSQL is a promising research direction, as it could enhance the ability of REDSQL to handle even larger databases in real-world applications.

5.4 Foundation Model Selection

Some LLM-based approaches are heavily dependent on the choice of the foundation model. For instance, a strategy that performs well with ChatGPT may not work with LLAMA3. In addition, developing REDSQL based on private LLMs can mitigate the risk of user data leakage. To investigate whether REDSQL is reliant on any specific foundational model, we evaluate its performance across a variety of foundation models. Table 5 displays the performance of REDSQL when implemented on different foundation models. In addition to ChatGPT, we include closed-source GPT4 and four open-sourced LLMs, LLAMA3, QWen, DeepSeek, and DeepSeekCoder.

The results show that REDSQL consistently enhances the performance of the baseline models across all mentioned LLMs, with improvements exceeding 4.5%. The open-sourced models can identify potential semantic errors and correct the SQL despite their limited capacity relative to the closed-source models.

Meanwhile, the open-sourced LLMs with about 70B parameters cannot achieve a performance similar to ChatGPT. This shows that REDSQL still relies on the capability of the foundation model to achieve higher accuracy. This reliance is acceptable, given that the refinement process requires LLMs to identify semantic errors and generate correct SQL. However, larger open-sourced models like DeepSeek and DeepSeekCoder achieve competitive performance compared to ChatGPT, suggesting that REDSQL can be deployed privately, **reducing the risk of user data leakage**.

Table 5: Different foundation model selection of REDSQL for accuracy (EX) evaluation based on Bird Dev.

LLM	CODES	PURPLE
Baseline	58.5	56.6
LLAMA3	63.6 (+5.1)	63.5 (+6.9)
QWen	63.1 (+4.6)	62.3 (+5.7)
DeepSeek	65.1 (+6.6)	64.0 (+7.4)
DeepSeekCoder	66.1 (+7.6)	64.3 (+7.7)
ChatGPT	66.2 (+7.7)	64.0 (+7.4)
GPT4	**67.3 (+8.8)**	**67.7 (+11.1)**

We notice that the LLMs all achieve high translation accuracy, suggesting that REDSQL does not depend on specific foundation model architectures. Notably, among models with similar architectures, such as DeepSeek and DeepSeekCoder, the code version of the model, which is specifically tuned for coding tasks, achieves higher accuracy. The finding is aligned with intuition, indicating that code-oriented LLMs are well-suited for REDSQL.

We incorporate GPT4 into REDSQL, leveraging its advanced capabilities to achieve the highest accuracy in our experiments. The integration of GPT4 not only underscores the adaptability of REDSQL to work with more advanced LLMs but also enhances its overall effectiveness. We think that REDSQL can consistently achieve higher performance with the development of LLMs. Unlike existing NL2SQL approaches simply based on the schema and partial data, REDSQL provides more comprehensive information about the database. REDSQL does not rely on the assumption that the LLMs can rightly infer the semantics of the database just based on the schema. We explicitly provide the data-related information in REDSQL, making it possible for LLMs to understand the database.

5.5 Ablation Study

We conduct an ablation study to evaluate the individual contributions of each REDSQL component. Specifically, we remove the documentation, SQL execution, context expansion, and constraints verification modules one at a time and measure the effects on accuracy. Additionally, we perform experiments where we integrate each REDSQL module sequentially, leaving only one module active at a time to refine the predicted SQL using REDSQL. The results of this study are presented in Table 6.

The ablation study results indicate that every component of REDSQL contributes to its overall performance, as the accuracy drops when any part is removed. Removing the documentation or SQL execution modules results in only minor performance declines. This suggests that while these modules provide supplementary information, they are not the core contributor to SQL refinement. This limited impact reflects their constrained ability to extract sufficient data, thereby failing to convey the true semantics of SQL fully.

The refinement performance of only integrating the constraints verification process is still remarkable, reflecting that the core idea of REDSQL in identifying potential errors by designed constraints on data is novel. Specifically, REDSQL enhances PURPLE accuracy by 6% solely through constraints verification, without leveraging an LLM ensemble effect (both utilize ChatGPT).

Table 6: Ablation Study.

Strategy	CODES	PURPLE
REDSQL	66.2	64.0
- Documentation	65.7 (-0.5)	63.7 (-0.3)
- SQL Execution	66.0 (-0.2)	63.0 (-1.0)
- Constraints Verification	64.0 (-2.2)	61.9 (-2.1)
- Context Expansion	64.8 (-1.4)	62.5 (-1.5)
Only Documentation	62.3 (-3.9)	58.9 (-5.1)
Only SQL Execution	62.8 (-3.4)	59.3 (-4.7)
Only Constraints Verification	64.3 (-1.9)	62.6 (-1.4)
Only Context Expansion	64.3 (-1.9)	61.2 (-2.8)

Context expansion enables LLMs to make judgments based on the violation report and within a more informative context, thus improving overall performance in SQL refinement. It is important to notice that the accuracy without context expansion is similar to the only constraints verification configuration. This indicates that the benefits of the documentation and SQL execution modules are fully realized only when integrated with the context expansion module. Providing additional schema information not referenced in the original SQL allows LLMs to refine based on the documentation or execution results effectively.

We notice that PURPLE suffers a more substantial accuracy drop when only one component is included, except when only constraints verification is active. This suggests that PURPLE relies heavily on the data context provided by constraints verification for SQL refinement. Since both REDSQL and PURPLE utilize ChatGPT, they require additional contextual information for effective refinement without relying on model-specific knowledge inputs. Although REDSQL does not incorporate prior knowledge of existing NL2SQL approaches, it would likely benefit from awareness of the foundational model initially used in the previous approach.

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed implementing constraints on query operations to evaluate their input relations for potential semantic error detection. Our framework verified the predicted SQL for correctness by identifying and reporting constraint violations. To further enhance SQL refinement, we enriched the SQL-related context for LLMs, providing more related information. Additionally, we introduced a documentation process that offers LLMs a global understanding of the database. REDSQL effectively addressed data integration challenges and improved the NL2SQL accuracy.

Two future work directions are identified: ① **Automatic query constraints discovery** could enable REDSQL to automatically adapt to specific workloads in databases without pre-defined constraints, though it remains challenging as the discovered constraints must align with business semantics. ② **Implementing REDSQL alongside the SQL execution process** could improve its effectiveness by leveraging existing DBMS optimization techniques, potentially reducing the latency of REDSQL.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 62272106).

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB Journal* 24, 4 (2015), 557–581.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data Profiling: A Tutorial. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*. 1747–1751.

[3] Alireza Ahadi, Julia Coleman Prior, Vahid Behbood, and Raymond Lister. 2016. Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. In *International Conference on Research and Development in Information Retrieval, SIGIR*. 272–277.

[4] Meike Albrecht, Edith Buchholz, Antje Düsterhöft, and Bernhard Thalheim. 1995. An Informal and Efficient Approach for Obtaining Semantic Constraints Using Sample Data and Natural Language Processing. In *International Conference on Machine Learning, ICML*, Vol. 1358. 1–28.

[5] Shengnan An, Bo Zhou, Zeqi Lin, Qiang Fu, Bei Chen, Nanning Zheng, Weizhu Chen, and Jian-Guang Lou. 2023. Skill-Based Few-Shot Selection for In-Context Learning. *arXiv* abs/2305.14210 (2023).

[6] William Ward Armstrong. 1974. Dependency Structures of Data Base Relationships. In *Applied Natural Language Processing Conference, ANLP*. 580–583.

[7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv* abs/2309.16609 (2023).

[8] Catriel Beeri and Philip A. Bernstein. 1979. Computational Problems Related to the Design of Normal Form Relational Schemas. *ACM Transactions on Database Systems* 4, 1 (1979), 30–59.

[9] Catriel Beeri and Moshe Y. Vardi. 1981. The Implication Problem for Data Dependencies. In *Human Language Technology*, Vol. 115. 73–85.

[10] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv* abs/2004.05150 (2020).

[11] Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Global Reasoning over Database Structures for Text-to-SQL Parsing. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*. 3657–3662.

[12] Stefan Brass and Christian Goldberg. 2005. Proving the safety of SQL queries. In *IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET)*. 197–204.

[13] Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software* 79, 5 (2006), 630–644.

[14] Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. 2021. LGESQL: Line Graph Enhanced Text-to-SQL Model with Mixed Local and Non-Local Relations. In *Workshop on Neural Machine Translation, NMT@ACL*. 2541–2555.

[15] Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2024. SQLFixAgent: Towards Semantic-Accurate SQL Generation via Multi-Agent Collaboration. *arXiv* (2024).

[16] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *VLDB Journal* 24, 6 (2015), 731–755.

[17] Shuaichen Chang and Eric Fosler-Lussier. 2023. How to Prompt LLMs for Text-to-SQL: A Study in Zero-shot, Single-domain, and Cross-domain Settings. *arXiv* abs/2305.11853 (2023).

[18] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *arXiv* abs/2304.05128 (2023).

[19] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. In *Proceedings of the VLDB Endowment*, Vol. 11. 1482–1495.

[20] E. F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *IBM Research Report* RJ987 (1972).

[21] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, Hao Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, Tao Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, and Xiaowen Sun. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *arXiv* abs/2405.04434 (2024).

[22] Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-Grounded Pretraining for Text-to-SQL. In *Frontiers in Education Conference, FIE*. 1337–1350.

[23] Benjamin Dietrich and Torsten Grust. 2015. A SQL Debugger Built from Spare Parts: Turning a SQL: 1999 Database System into Its Own Debugger. In *Proceedings of the ACM on Management of Data, SIGMOD*. 865–870.

[24] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. *arXiv* abs/2307.07306 (2023).

[25] Ronald Fagin. 1977. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems* 2, 3 (1977), 262–278.

[26] Ronald Fagin. 1981. A Normal Form for Relational Databases That Is Based on Domians and Keys. *ACM Transactions on Database Systems* 6, 3 (1981), 387–415.

[27] Ronald Fagin. 1982. Horn clauses and database dependencies. *J. ACM* 29, 4 (1982), 952–985.

[28] Yuankai Fan, Zhenying He, Tonghui Ren, Dianjun Guo, Lin Chen, Ruisi Zhu, Guanduo Chen, Yinan Jing, Kai Zhang, and X. Sean Wang. 2023. Gar: A Generate-and-Rank Approach for Natural Language to SQL Translation. In *International Conference on Intellectual Systems and Computer Science*. 110–122.

[29] Yuankai Fan, Zhenying He, Tonghui Ren, Can Huang, Yinan Jing, Kai Zhang, and X. Sean Wang. 2024. Metasql: A Generate-Then-Rank Framework for Natural Language to SQL Translation. In *International Conference on Data Engineering ,ICDE*. 1765–1778.

[30] Yuankai Fan, Tonghui Ren, Zhenying He, X. Sean Wang, Ye Zhang, and Xingang Li. 2023. GenSql: A Generative Natural Language Interface to Database Systems. In *International Conference on Data Engineering ,ICDE*. 3603–3606.

[31] Yuankai Fan, Tonghui Ren, Can Huang, Zhenying He, and X. Sean Wang. 2024. Grounding Natural Language to SQL Translation with Data-Based Self-Explanations. *arXiv* abs/2411.02948 (2024).

[32] Yuankai Fan, Tonghui Ren, Can Huang, Beini Zheng, Yinan Jing, Zhenying He, Jinbao Li, and Jianxin Li. 2024. A confidence-based knowledge integration framework for cross-domain table question answering. *Knowledg-Based Systems* 306 (2024), 112718.

[33] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. In *Proceedings of the VLDB Endowment*, Vol. 16. 1534–1547.

[34] Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021. Towards Robustness of Text-to-SQL Models against Synonym Substitution. In *Workshop on Teaching, Learning and Assessment in Databases*. 2505–2515.

[35] Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring Underexplored Limitations of Cross-Domain Text-to-SQL Generalization. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*. 8926–8931.

[36] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. In *Proceedings of the VLDB Endowment*, Vol. 17. 1132–1145.

[37] Amir Gilad, Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2022. Understanding Queries by Conditional Instances. In *Conference on Empirical Methods in Natural Language Processing, EMNLP*. 355–368.

[38] Orest Gkini, Theofilos Belmpas, Georgia Koutrika, and Yannis E. Ioannidis. 2021. An In-Depth Benchmarking of Text-to-SQL Systems. In *Text REtrieval Conference, TREC*. 632–644.

[39] Christian Goldberg. 2009. Do you know SQL? About semantic errors in database queries. In *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing, ACL/IJCNLP*.

[40] Torsten Grust and Jan Rittinger. 2013. Observing SQL queries in their natural habitat. *ACM Transactions on Database Systems* 38, 1 (2013), 3.

[41] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot Text-to-SQL Translation using Structure and Content Prompt Learning. *Proceedings of the ACM on Management of Data, SIGMOD* 1, 2 (2023), 147:1–147:28.

[42] Zihui Gu, Ju Fan, Nan Tang, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Sam Madden, and Xiaoyong Du. 2023. Interleaving Pre-Trained Language Models and Large Language Models for Zero-Shot NL2SQL Generation. *arXiv* abs/2306.08891 (2023).

[43] Yihao Hu, Amir Gilad, Kristin Stephens-Martinez, Sudeepa Roy, and Jun Yang. 2024. Qr-Hint: Actionable Hints Towards Correcting Wrong SQL Queries. *Proceedings of the ACM on Management of Data, SIGMOD* 2, 3 (2024), 164.

[44] Binyuan Hui, Ruiying Geng, Lihan Wang, Bowen Qin, Yanyang Li, Bowen Li, Jian Sun, and Yongbin Li. 2022. S^2SQL: Injecting Syntax to Question-Schema Interaction Graph Encoder for Text-to-SQL Parsers. In *Proceedings of the VLDB Endowment*. 1254–1262.

[45] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today?. In *Proceedings of the VLDB Endowment*, Vol. 13. 1737–1750.

[46] Anthony C. Klug. 1982. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *J. ACM* 29, 3 (1982), 699–717.

[47] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. MCS-SQL: Leveraging Multiple Prompts and Multiple-Choice Selection For Text-to-SQL Generation. In *International Conference on Computational Linguistics, COLING*. 337–353.

[48] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, H. V. Jagadish, and Mirek Riedewald. 2020. QueryVis: Logic-based Diagrams help Users Understand Complicated SQL Queries Faster. In *Findings of the Association for Computational Linguistics, ACL*. 2303–2318.

[49] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? [Experiment, Analysis & Benchmark]. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3318–3331.

[50] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. In *Proceedings of the VLDB Endowment*, Vol. 8. 73–84.

[51] Fei Li and H. V. Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *Findings of the Association for Computational Linguistics, EMNLP*. 709–712.

[52] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *Advances in Neural Information Processing Systems, NeurIPS*. 13067–13075.

[53] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. In *Proceedings of the ACM on Management of Data, SIGMOD*, Vol. 2. 127.

[54] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-T5: Mixing Pre-trained Transformers with Graph-Aware Layers for Text-to-SQL Parsing. In *Proceedings of the ACM on Management of Data, SIGMOD*. 13076–13084.

[55] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems, NeurIPS*.

[56] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *arXiv* abs/2305.06161 (2023).

[57] Zhishuai Li, Xiang Wang, Jingjing Zhao, Sun Yang, Guoqing Du, Xiaoru Hu, Bin Zhang, Yuxiao Ye, Ziyue Li, Rui Zhao, and Hangyu Mao. 2024. PET-SQL: A Prompt-enhanced Two-stage Text-to-SQL Framework with Cross-consistency. *arXiv* abs/2403.09732 (2024).

[58] Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing. In *Proceedings of the VLDB Endowment*, Vol. EMNLP 2020. 4870–4888.

[59] Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S. Yu. 2023. A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability. *arXiv* abs/2303.13547 (2023).

[60] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *arXiv* abs/2408.05109 (2024).

[61] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*. 503–520.

[62] Felix Naumann. 2013. Data profiling revisited. In *Proceedings of the ACM on Management of Data, SIGMOD*, Vol. 42. 40–49.

[63] Gultekin Özsoyoglu, Z. Meral Özsoyoglu, and Victor Matos. 1987. Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions. *ACM Transactions on Database Systems* 12, 4 (1987), 566–592.

[64] Gultekin Özsoyoglu and Huaqing Wang. 1989. A Relational Calculus with Set Operators, Its Safety and Equivalent Graphical Languages. *IEEE Transactions on Software Engineering* 15, 9 (1989), 1038–1052.

[65] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *Conference on Innovation and Technology in Computer Science Education, ITiCSE*.

[66] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Advances in Neural Information Processing Systems, NeurIPS*.

[67] Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. In *Findings of the Association for Computational Linguistics, EMNLP*. 8212–8220.

[68] Kai Presler-Marshall, Sarah Heckman, and Kathryn T. Stolee. 2021. SQLRepair: Identifying and Repairing Mistakes in Student-Authored SQL Queries. In *Findings of the Association for Computational Linguistics, ACL*. 199–210.

[69] Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. RASAT: Integrating Relational Structures into Pretrained Seq2Seq Model for Text-to-SQL. In *International Semantic Web Conference and Asian Semantic Web Conference, ISWC + ASWC*. 3215–3229.

[70] Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before Generation, Align it! A Novel and Effective Strategy for Mitigating Hallucinations in Text-to-SQL Generation. In *Findings of the Association for Computational Linguistics, ACL*. Association for Computational Linguistics, 5456–5471.

[71] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 140:1–140:67.

[72] Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and X. Sean Wang. 2024. PURPLE: Making a Large Language Model a Better SQL Writer. In *International Conference on Data Engineering ,ICDE*. 15–28.

[73] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. 1988. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems* 13, 4 (1988), 389–417.

[74] Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. In *Findings of the Association for Computational Linguistics, EMNLP*. 12–21.

[75] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. In *Proceedings of the VLDB Endowment*, Vol. 9. 1209–1220.

[76] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Networks* 20, 1 (2009), 61–80.

[77] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Conference on Empirical Methods in Natural Language Processing, EMNLP*. 9895–9901.

[78] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish R. Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++: Natural Language Querying for Complex Nested SQL Queries. In *Proceedings of the VLDB Endowment*, Vol. 13. 2747–2759.

[79] Alkis Simitsis, Georgia Koutrika, and Yannis E. Ioannidis. 2008. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB Journal* 17, 1 (2008), 117–149.

[80] John B. Smelcer. 1995. User errors in database query composition. *International Journal of Human-Computer Studies* 42, 4 (1995), 353–381.

[81] Toni Taipalus. 2020. Explaining Causes Behind SQL Query Formulation Errors. In *Text REtrieval Conference, TREC*. 1–9.

[82] Toni Taipalus, Mikko T. Siponen, and Tero Vartiainen. 2018. Errors and Complications in SQL Query Formulation. *ACM Transactions on Computing Education* 18, 3 (2018), 15:1–15:29.

[83] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. *arXiv preprint arXiv:2405.16755* (2024).

[84] Bernhard Thalheim. 1996. An overview on semantical constraints for database models. In *Automata, Languages and Programming*.

[85] Dayton G. Thorpe, Andrew J. Duberstein, and Ian A. Kinsey. 2024. Dubo-SQL: Diverse Retrieval-Augmented Generation and Fine Tuning for Text-to-SQL. *arXiv* abs/2404.12560 (2024).

[86] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *International Conference on Learning Representations, ICLR*. 137–146.

[87] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *Proceedings of the ACM on Management of Data, SIGMOD*, Vol. 6355. 425–446.

[88] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *International Conference on Computational Linguistics, COLING*. 540–557.

[89] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *International Conference on Data Engineering ,ICDE*. 7567–7578.

[90] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Conference on Innovative Data Systems Research, CIDR*. 3911–3921.

[91] John M. Zelle and Raymond J. Mooney. 1996. Learning to Parse Database Queries Using Inductive Logic Programming. In *Advances in Neural Information Processing Systems, NeurIPS*. 1050–1055.

[92] Lu Zeng, Sree Hari Krishnan Parthasarathi, and Dilek Hakkani-Tur. 2022. N-Best Hypotheses Reranking for Text-to-SQL Systems. In *International Conference on Learning Representations, ICLR*. 663–670.

[93] Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Sun Yang, Chi Harold Liu, Rui Zhao, Ziyue Li, and Hangyu Mao. 2024. Benchmarking the Text-to-SQL Capability of Large Language Models: A Comprehensive Evaluation. *arXiv* abs/2403.02951 (2024).

[94] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. In *Findings of the Association for Computational Linguistics, EMNLP*. 93–105.

[95] Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. 2023. ScienceBenchmark: A Complex Real-World Benchmark for Evaluating Natural Language to SQL Systems. In *Proceedings of the VLDB Endowment*, Vol. 17. 685–698.

[96] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv* (2024).