Database-Centric NL2SQL

Anonymous ACL submission

Abstract

Existing NL2SQL systems rely heavily on LLMs, prompting them with database schema descriptions. However, real-world databases often contain complex schemas, ambiguous naming, and schema-incompliant instances, making accurate SQL generation challenging. Additionally, recent trends of using long prompts and generating multiple candidate queries contribute to high computational costs.

To mitigate these issues, we propose two *Database-Centric techniques*: *View-based Optimization*, which simplifies schema representation using database views, and *Database-as-a-Tool*, which leverages database functionalities to refine SQL queries.[1] Our approach achieves an execution accuracy of 70.47% on the BIRD benchmark, comparable to existing NL2SQL methods, while greatly reducing input tokens by 17× to 374×.

1 Introduction

Interacting with databases through natural language marks a significant paradigm shift in how users access and manage data. Natural Language to SQL (NL2SQL), which translates natural language questions into SQL queries, democratizes non-expert users to query and manipulate databases using everyday language. This innovation has significant potential for business applications by empowering stakeholders to analyze data and derive actionable insights. The rise of large language models (LLMs) has greatly advanced NL2SQL frameworks, improving execution accuracy by leveraging contextual understanding to bridge the gap between natural language questions and SQL queries (Gao et al., 2024b; Pourreza et al., 2024; Maamari et al., 2024; Talaei et al., 2024; Pourreza and Rafiei, 2024; Gao et al., 2024a; Zhang et al., 2023a).

Existing LLM-empowered NL2SQL frameworks typically follow a structured pipeline comprising three stages: schema linking, SQL generation, and refinement. Schema linking maps key terms from a natural language question to relevant database components, such as tables and columns. SQL generation then translates the natural language question into a syntactically and semantically valid SQL statement. Finally, the refinement stage corrects errors to ensure that the generated SQL statement is executable and aligns with the user's intent.

Despite significant progress, adapting NL2SQL frameworks for real-world applications presents several limitations. **(L1)** One key issue is the complexity of database schemas (Floratou et al., 2024; Li et al., 2024a; Zhang et al., 2024b), which increases the amount of information LLMs must process when generating SQL queries. While detailed table and column descriptions aid comprehension, large schemas with numerous interrelationships amplify computational overhead. This not only heightens computational overhead but also makes schema processing more resource-intensive. Thus, an alternative representation is required to present schemas in a more structured and manageable way. **(L2)** Moreover, while recent LLMs excel in reasoning based on the given prompt and pre-trained knowledge, they overlook real-world database structures and therefore often generate incorrect SQL queries. Specifically, LLMs cannot fully comprehend the structure of an actual database, as its concrete properties cannot be inferred solely from trivial schema information (e.g., table and column names, descriptions, and sample values). For instance, they may struggle to infer one-to-one or one-to-many relationships between columns based solely on their names. Consequently, leveraging external tools to extract more information from the database itself becomes essential.

To address these limitations, we propose Database-Centric NL2SQL, which systematically integrates database functionalities into the NL2SQL process (Figure 1). This method offers

[1]The source code will be released after publication.

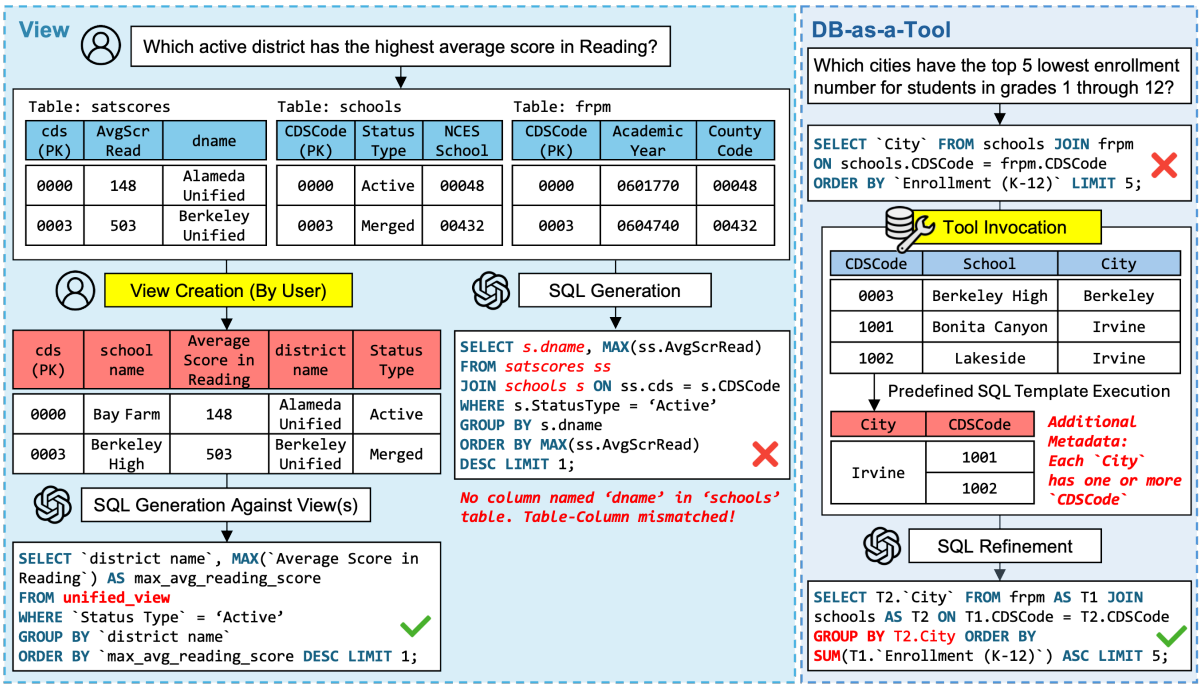


Figure 1: Overview of Databse-Centric NL2SQL.

two main advantages. First, it alleviates schema complexity by utilizing database views to provide a refined schema representation. Second, it refines SQL queries by extracting metadata from the database, enabling the LLMs to better comprehend the database's underlying structure and its data. Database-Centric NL2SQL aims to overcome the limitations of relying solely on LLMs and improve their reliability for real-world applications.

We outline the key challenges in integrating database functionalities into the NL2SQL framework and present our solutions as follows.

C1: How to resolve schema complexity? The complexity of large database schemas makes schema interpretation difficult, as LLMs struggle to process extensive and interconnected structures. Column names may include abbreviations or naming variations, and different tables can contain overlapping column names, making it difficult to determine the correct mapping. To mitigate this, detailed descriptions of tables and columns are often included in prompts to enhance schema comprehension. However, this increases prompt length and computational cost, limiting scalability. Our proposed solution is to leverage "**views**", which restructure the schema into a more interpretable format. Views clarify ambiguous table and column names, group tables, and encapsulate frequently used SQL query patterns (Halevy, 2001). By reducing schema complexity, views allow LLMs to focus on generating accurate SQL queries rather than navigating complex structures.

C2: How can LLMs receive additional information to better understand the database structure? To overcome the limitations of LLMs utilizing only a limited scope of database information, external tools beyond LLMs are required (Li et al., 2023; Qin et al., 2024). In the NL2SQL task, we identify the database itself as a suitable tool for this purpose and introduce **Database-as-a-Tool (DBTool)**, a tool-augmented NL2SQL approach that enables LLMs to interact dynamically with databases. Similar to how database practitioners manually explore databases by running queries and analyzing results, DBTool automates database exploration by querying the database to extract metadata, which provides essential contextual information for refining erroneous SQL queries.

To validate the effectiveness of our Database-Centric NL2SQL approaches, we conducted evaluations on the development set of the BIRD benchmark (Li et al., 2024b). Our proposed method achieves an execution accuracy of 70.47%, while significantly reducing the number of input tokens, ranging from a 17x to 374x reduction compared to other methods. This indicates that current NL2SQL systems largely based on prompt engineering, have room for improvement in token efficiency by leveraging the database.

2 Related Work

2.1 Existing NL2SQL Frameworks

Schema Linking. Schema linking aligns natural language questions with the relevant tables and columns in a database schema. Prior research has introduced techniques to address the accuracy of schema linking. One prominent approach is schema pruning, which removes irrelevant tables and columns to reduce the amount of information LLMs must process (Lei et al., 2020; Pourreza and Rafiei, 2024; Xie et al., 2024; Glenn et al., 2023; Gao et al., 2024b; Caferoğlu and Ulusoy, 2024). Another line of work focuses on enriching schema representations by incorporating detailed descriptions and sample values, enabling LLMs to better infer contextual information and produce accurate SQLs (Talaei et al., 2024; Pourreza et al., 2024; Li et al., 2024a; Zhu et al., 2024).

SQL Generation and Refinement. The SQL generation stage often leverages advanced capabilities of LLMs, such as few-shot learning (Brown et al., 2020; Dong et al., 2023; Gao et al., 2024a; Nan et al., 2023) and chain-of-thought reasoning (Wei et al., 2022; Tai et al., 2023). Furthermore, recent research has increasingly adopted a strategy where the model generates multiple SQL query candidates and selects the most accurate one (Gao et al., 2024b; Pourreza et al., 2024; Talaei et al., 2024; Lee et al., 2024).

Meanwhile, the SQL refinement stage addresses syntactic and logical errors in the SQL query. This process involves re-generating SQL queries based on execution results or validation guidelines (Pourreza and Rafiei, 2024; Talaei et al., 2024; Wang et al., 2023; Cen et al., 2024; Ren et al., 2024).

2.2 Database-Centric NLSQL

View-based NL2SQL. While existing schema-linking methods focus on pruning and organizing schema details, they do not fundamentally restructure the schema. As a result, large schemas must still be fully considered during query generation. In contrast, every database system supports views to encapsulate data or present simplified schema representations, abstracting schema complexity from end-users (Halevy, 2001). However, few studies have explored leveraging database views to simplify schema complexity in NL2SQL.

Tool-augmented NL2SQL. Tool-augmented generation is an emerging technique designed to enhance the accuracy and reliability of LLM responses by training or prompting LLMs to utilize external tools (Parisi et al., 2022; Hsieh et al., 2023; Paranjape et al., 2023; Hao et al., 2023; Qiao et al., 2023). Tool-augmented models can invoke tools such as search engines, or calculators to perform operations beyond their internal reasoning capabilities (Schick et al., 2024; Li et al., 2023; Qin et al., 2024; Mialon et al., 2023; Nakano et al., 2021). This approach has proven effective in a range of tasks, including question answering, mathematical problem solving, and code generation (Lu et al., 2024; Zhang et al., 2024a, 2023b; Zhuang et al., 2023). However, tool-augmented generation remains unexplored for NL2SQL, presenting an opportunity to improve the accuracy of NL2SQL.

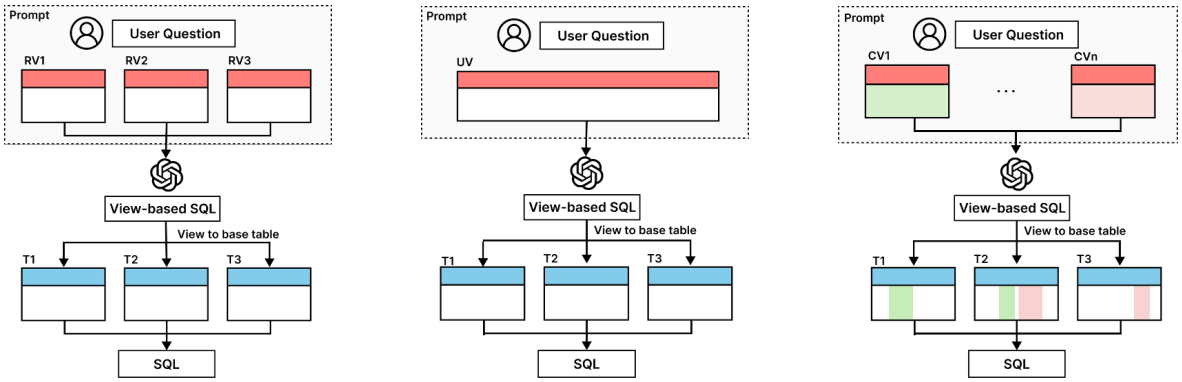
3 View-based NL2SQL

The following sections introduce **Database-Centric NL2SQL**, which comprises *View-based NL2SQL* and *Tool-augmented NL2SQL*. View-based NL2SQL mitigates schema complexity by reorganizing schema structures into more interpretable representations, while Tool-augmented NL2SQL provides LLMs with additional database-specific metadata, thereby enhancing schema understanding by incorporating the database as a tool.

View-based NL2SQL reformulates database schemas into representations optimized for LLM interpretation. Instead of exposing raw schema structures of base tables, we introduce three types of views: *Renamed Views*, *Unified Views*, and *Customized Views*. As shown on left side of Figure 1, users need to create views, and the LLM then generates SQL query based on these views. Figure 2 illustrates how each of three view types is processed with a user question. Each view is incorporated into the prompt along with the user question, but only the view schema is provided rather than an extensive description. Once the LLM generates an SQL query against the view-based schema representation, an internal rewriting algorithm (Appendix B) transforms it into a SQL query that directly references the original base tables. In the following, we discuss the three types of views we use.

3.1 Renamed Views

Renamed Views enhance schema clarity by resolving column name ambiguity. Unlike existing schema-linking methods that infer column semantics from descriptions, they explicitly redefine column names, ensuring precise identification and



(a) Renamed View (RV). (b) Unified View (UV). (c) Customized View (CV).

Figure 2: View-based NL2SQL, illustrating how each of the three view types is used to handle user query.

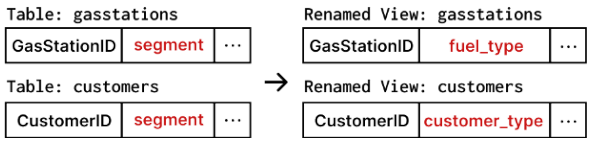


Figure 3: Example of Renamed View.

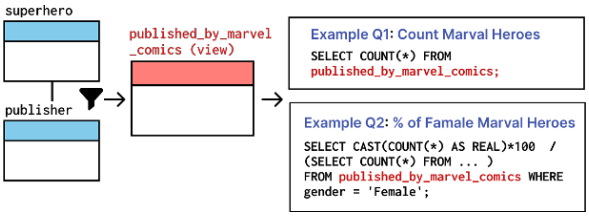


Figure 4: Example of Customized View.

minimizing misinterpretation. Renamed Views adopt detailed names, including data type if necessary. Because database views are widely used in enterprise environments, adopting Renamed Views are expected to incur minimal overhead. Figure 3 illustrates an example of Renamed Views: the 'segment' column in the 'customers' table is renamed to 'customer_type' and in the 'gasstations' table, it is renamed to 'fuel_type'. This transformation ensures explicit semantic differentiation, removing ambiguity and improving interpretability for LLMs.

3.2 Unified Views

As database schemas expand with more tables and columns, LLMs struggle to accurately associate columns with their respective tables due to complex schemas. A Unified View mitigates this issue by creating a virtual schema that integrates related tables into a single virtual representation, eliminating the need for LLMs to infer implicit joins. To minimize inefficiencies, Unified Views are selectively applied (discussed in Section 3.4) only when the generated SQL query fails due to incorrect table-column associations. Figure 1 presents an example of Unified View, where column 'dname' was incorrectly mapped to the table 'schools'. Provided with Unified Views, the model can generate SQL query with precise table-column matching without inferring joins.

3.3 Customized Views

Customized Views improve query execution by converting frequently used query patterns into predefined views, allowing direct access to intermediate results. By reducing redundant filtering and aggregation, these views simplify query formulation and lower processing overhead. Figure 4 illustrates how predefined views can streamline queries in superhero database (Li et al., 2024b). When natural language questions frequently involve Marvel superheroes, a predefined Marvel-Comics view can be created to store relevant data as an intermediate representation. This allows queries to directly reference the view instead of repeatedly filtering for Marvel-related attributes.

3.4 View Selection

To automate view selection, we employ a view selection algorithm under the assumption that all views are manually created in advance. In this study, each database includes one Renamed View per table, one Combined View, and approximately ten predefined Customized Views, ensuring flexibility in query transformation.

First, the user question is analyzed to determine whether it requires formulas or aggregation functions using an LLM-based prediction step. If not, a Renamed View is used; otherwise, a Customized

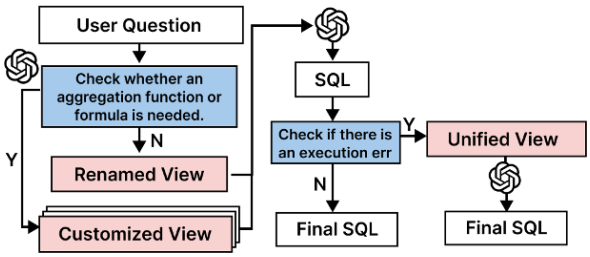


Figure 5: Workflow of view selection.

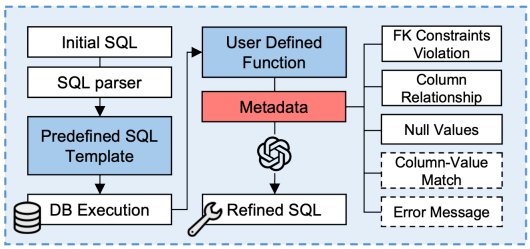


Figure 6: Overall flow of Database-as-a-Tool.

```
SELECT {from_tbl}.{from_col} FROM {from_tbl}
LEFT JOIN {to_tbl} ON {from_tbl}.{
from_col} = {to_tbl}.{to_col}
WHERE {to_tbl}.{to_col} IS NULL
```

Figure 7: Example of a predefined SQL template (violation of FK constraints).

```
def is_fk_violation(from_tbl.fk, to_tbl.pk,
    exec_result):
   if exec_result: # If the result is not
       empty, violation exists
      return (f"Foreign key constraint
          violation detected: Some values in
           '{from_tbl.fk}' do not match any
          value in '{from_tbl.pk}'.")
   return None
```

Figure 8: Example of user-defined function for converting execution result into metadata (violation of FK constraints).

View is retrieved. The selection process employs a similarity search mechanism that matches the user question against stored view definitions using Chroma database (Chroma-core, 2025), selecting up to three candidate views. When a Customized View is selected, its definition is embedded into the prompt alongside the Renamed View definition, allowing the LLM to reference the overall schema structure. After the LLM generates an SQL query, a post-processing step verifies whether an execution error occurs. If an error related to column name ambiguity is detected, it indicates an incorrect column-to-table mapping. In such cases, the Unified View definition is incorporated into the prompt, and the SQL query is regenerated to resolve the ambiguity. Detailed prompts are provided in Appendix B.

4 Tool-augmented NL2SQL

Database-as-a-Tool (DBTool) is the novel tool-augmented NL2SQL technique that directly extracts key metadata from the database for query refinement. We currently support three types of metadata about the underlying database instances: (1) *foreign key (FK) constraints violations*, (2) *one-to-many or one-to-one relationships between columns*, and (3) *the presence of null values in columns*. With such metadata, LLM can rectify some errors in its initial SQL query.

4.1 Overall Flow

DBTool follows a structured workflow, which is illustrated in Figure 6. When DBTool is invoked under specific conditions related to the database or the initial SQL query, as detailed in Appendix C, it extracts the column and table names from the initial query using an SQL query parser (e.g., sql-metadata (Brencz, 2024), sqlglot (Mao, 2023)). The query parser breaks down a query into its components by performing token analysis and constructing an abstract syntax tree (AST).

Then, the extracted table and column names are filled into predefined SQL templates that are designed to extract specific information about the database (e.g., Figure 7). Each SQL template is populated with parameters and then run against the database. Its result is then converted into metadata in a natural language format that can be easily interpreted by the LLM, through a corresponding user-defined function (UDF; e.g., Figure 8).

Finally, the metadata is incorporated into the prompt to generate a refined SQL, along with the metadata-specific guidelines. Although all three types of metadata are extracted from the database through the same flow, each type has a distinct predefined SQL template and UDF for converting results into metadata.

The three main types of metadata extracted by DBTool are described in the following sections. A step-by-step explanation of each process in DBTool is also provided in Appendix C for further clarity.

Q: What is the phone number of the school that has the highest average score in Math?

GPT-Generated SQL:

```
SELECT T2.Phone FROM satscores AS T1 INNER JOIN
schools AS T2 ON T1.cds = T2.CDSCode
WHERE T1.AvgScrMath = (SELECT MAX(AvgScrMath)
FROM satscores) LIMIT 1;
```

DBTool-refined SQL:

```
SELECT T2.Phone FROM satscores AS INNER JOIN
T1 schools AS T2 ON T1.cds = T2.CDSCode
WHERE T1.AvgScrMath = (SELECT MAX(AvgScrMath)
FROM satscores AS T1 INNER JOIN schools AS T2
ON T1.cds = T2.CDSCode) LIMIT 1;
```

Table 1: DBTool query refinement example: using metadata (FK constraint violation).

Q: Which cities have the top 5 lowest enrollment number for students in grades 1 through 12?

GPT-Generated SQL:

```
SELECT City FROM schools JOIN satscores
ON frpm.CDSCode = schools.CDSCode
ORDER BY `Enrollment (K-12)` ASC LIMIT 5;
```

DBTool-refined SQL:

```
SELECT City FROM schools JOIN satscores
ON frpm.CDSCode = schools.CDSCode
GROUP BY City
ORDER BY SUM(`Enrollment (K-12)`) ASC LIMIT 5;
```

Table 2: DBTool query refinement example: using metadata (one-to-many column relationship).

4.2 Violation of Foreign Key Constraints

In well-structured databases, an FK column in a *referencing* table is expected to match the corresponding primary key (PK) column in the *referenced* table, ensuring data consistency. However, in real-world databases, this consistency is often compromised, as FK constraints may not always be strictly enforced due to database configuration or domain-specific requirements (SQLite, 2024; PlanetScale, 2024). Detecting FK constraints violations is particularly crucial when joining tables or creating subqueries, as such violations can lead to an unexpected results such as missing records.

For example, in Table 1, the initial query attempted to find the phone number of the school with the highest math score by joining the 'satscores' (referencing table) and 'schools' (referenced table) tables. However, this query returned no records because the school with the highest score in 'satscores', identified by 'CDSCode' or 'cds', did not have a corresponding entry in the 'schools' table due to an FK violation. The tool detected this violation and returned metadata indicating an FK violation, using the metadata-specific SQL template (Figure 7) and UDF (Figure 8). Based on the metadata, the subquery is reconstructed to find the phone number of the highest-scoring school that exists in both tables. This adjustment ensures that the refined query produces valid results despite data inconsistencies.

4.3 Column Relationships

Understanding relationships between columns, such as one-to-many or one-to-one, is essential for accurate query formulation, especially when GROUP BY clause and aggregate functions are involved. Table 2 illustrates an example. The initially generated query did not include GROUP BY needed to aggregate data by city and sum student enrollments for each city. The tool identified a one-to-many relationship between 'City' and 'CDSCode', where multiple 'CDSCode' values correspond to a single 'City'. As a result, the query was corrected by adding GROUP BY with 'City' and using the SUM aggregate function, ensuring it aligns with the user's intent. See Figures 22 and 24 for the SQL template and UDF in this example.

4.4 Presence of Null Values

Verifying the presence of null values, which are especially common in real-world databases, is crucial for generating both user-friendly and accurate queries. The tool prevents unnecessary null values from being returned to the user by identifying columns that contain null values and instructing the LLM to apply the IS NOT NULL condition in SQL queries to filter them out. At the same time, it mitigates potential issues in scenarios where the COUNT function, ORDER BY clause, or arithmetic operations are used in SQL queries, as null values can lead to unexpected results such as incorrect counts, unintended sorting behavior, or computational errors. Figure 23 and 25 illustrate the SQL template and the UDF for extracting the metadata.

4.5 Handling Other Errors with DBTool

Furthermore, DBTool efficiently corrects column-value mismatches and execution errors using fewer tokens than existing methods, which handle these errors with lengthy prompts, leading to high inference costs. These functionalities operate differently from the three metadata extraction processes described earlier and are detailed in Appendix C.

Column-value mismatch error occurs when a

Method	EX (%)
CHASE-SQL+Gemini (Pourreza et al., 2024)	74.46
XiYan-SQL (Gao et al., 2024b)	73.34
$\text{CHESS}_{\text{IR+CG+UT}}$ (Talaei et al., 2024)	68.31
Distillery (Maamari et al., 2024)	67.21
$\text{XiYan-SQL}_{\text{QwenCoder-32B}}$ (Gao et al., 2024b)	67.01
$\text{CHESS}_{\text{IR+SS+CG}}$ (Talaei et al., 2024)	65.00
E-SQL (Caferoğlu and Ulusoy, 2024)	65.58
RSL-SQL+DeepSeek (Cao et al., 2024)	63.56
MCS-SQL+GPT-4 (Lee et al., 2024)	63.36
Baseline	57.30
Baseline w/ desc.	60.63
View (Ours)	65.26
DBTool (Ours)	63.03
View+DBTool (Ours)	**70.47**

Table 3: Comparison with other NL2SQL systems (BIRD-SQL's dev set (Li et al., 2024b)). Our methods and Baseline configurations used GPT-4o.

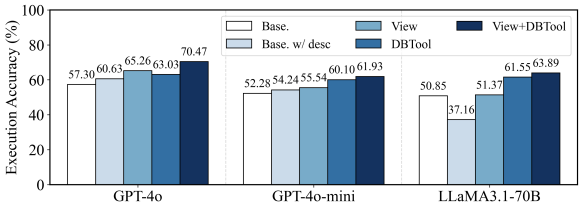


Figure 9: Execution accuracy (BIRD-SQL's dev set).

WHERE clause in an SQL query references non-existent values, resulting in no records being returned from the database. DBTool utilizes an SQL query parser to extract the column referenced in the WHERE clause and searches for relevant values that align with the user's question. **Execution Error** occurs when a query fails to execute at runtime. DBTool analyzes the error message and parses the SQL query to extract only the relevant tables and columns necessary for debugging.

5 Performance Evaluation

5.1 Setting

Dataset. Performance evaluation was conducted using the BIRD benchmark (Li et al., 2024b), a cross-domain dataset designed for NL2SQL task. BIRD includes natural language questions, corresponding ground-truth SQL queries (gold SQLs), and metadata such as table/column descriptions and knowledge evidence. All experiments were performed on the development set (dev set).

Metric. The primary evaluation metric, Execution Accuracy (EX, %), measures the proportion of generated SQL queries that produce execution results identical to those of the ground-truth SQL queries. Additionally, inference cost, measured by token count, was analyzed for experiments conducted with a closed-source LLM.

LLMs. The proposed methods were evaluated across three LLMs, including closed-source models (OpenAI GPT-4o, GPT-4o-mini (Achiam et al., 2023)) and open-source models (LLaMA 3.1-70B (Dubey et al., 2024)). The temperature of all LLMs was set to 0 to ensure deterministic output.

Baseline. The baseline is based on the OpenAI demonstration prompt, which was first introduced in OpenAI's official Text-to-SQL demo (Gao et al., 2024a). Another baseline, referred to as *baseline with description (baseline w/ desc)*, incorporates table and column descriptions (Maamari et al., 2024) provided by the BIRD benchmark (Li et al., 2024b). The specific prompts for *baseline* and *baseline w/ desc* are included in Appendix A.

5.2 Execution Accuracy

As summarized in Table 3, our experiments demonstrate that *View+DBTool* achieves an execution accuracy of 70.47%, comparable to state-of-the-art methods. Figure 9 further illustrates the execution accuracy across different LLMs, comparing our methods with the *baseline* and *baseline w/ desc*. Notably, *view* consistently outperforms the *baseline* and shows an improvement over *baseline w/ desc* across all three LLMs, with the most pronounced gains observed on LLaMA. In addition, applying *DBTool* to SQL queries initially generated incorrectly by *baseline* or *view*, improves performance by approximately 5% to 12%. Once again, the largest benefit appears in experiments with LLaMA. Collectively, these results highlight the effectiveness and stability of our NL2SQL approach when deployed with different LLMs.

5.3 Cost Efficiency

As shown in Figure 10, token consumption in the *baseline w/ desc* increases steeply with the number of columns, whereas *View+DBTool* maintains consistently low usage. This contrast underscores how embedding extensive schema descriptions in prompts becomes inefficient, as the associated inference cost scales disproportionately with database size, making them impractical for large-scale applications. Other methods handle most of their schema reasoning directly within prompts, leading to higher token counts, whereas *View+DBTool* offloads these operations to the database side, thereby reducing the burden on the

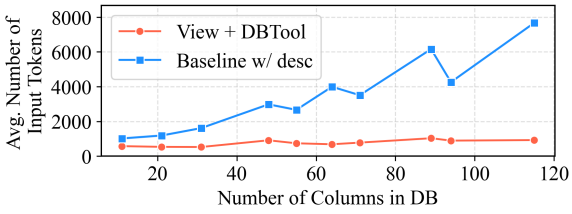


Figure 10: Average number of input tokens per query vs. the number of columnsthe database (GPT-4o-mini, BIRD-SQL's dev set).

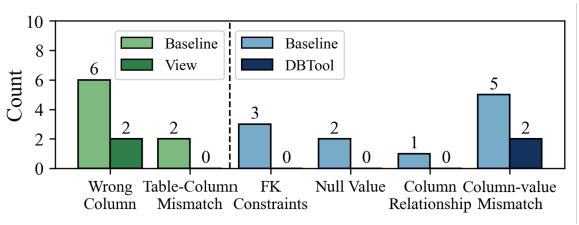


Figure 11: Distribution of errors (GPT-4o).

Method	Input (K)	Output (K)	EX (%)
CHESS$_{IR+CG+UT}$ *	307.51	25.26	68.31
RSL-SQL†	14.28	0.48	67.21
E-SQL‡	26.24	0.80	65.58
View + DBTool	**0.82**	**0.15**	**70.47**
Baseline (w/ desc).	3.63	0.05	60.63

Table 4: Average number of tokens per query (BIRD's dev set). Results are from *an actual experiment using 147 subsamples provided by the author, †the original paper, and ‡GitHub.

prompts themselves.

Table 4 compares the average number of tokens required per query across existing NL2SQL methods. *View+DBTool* achieves 70.47% execution accuracy while consuming only 0.82K input tokens, representing a 4.42× reduction compared to the *baseline*. It also reduces input token consumption by a factor of 17× to 374× compared to other methods whose code is publicly available in github, while offering similar performance. Although *View+DBTool* generates slightly more output tokens than the *baseline* due to CoT prompting, it still produces 4 to 20 times fewer output tokens than other methods. This substantial reduction in token consumption highlights the token efficiency of our approach, without sacrificing accuracy and reducing reliance on lengthy prompts. Since token usage directly drives costs in closed LLMs and adds computational overhead in open-source models, *View+DBTool* offers a scalable, cost-effective solution for SQL generation in real-world databases.

5.4 Error Analysis

This section evaluates the effectiveness of each method by analyzing error reduction in the "California Schools" database from the BIRD Benchmark.

Figure 11 demonstrates how *view* effectively mitigates schema-related errors, addressing the majority of identified issues. Specifically, Renamed Views reduce wrong-column errors by standardizing ambiguous column names, thereby enhancing clarity. Unified Views resolve table-column mismatch errors by pre-joining table-column associations. On the other hand, Customized Views target query efficiency by serving as intermediate representations for frequently used query patterns and thus lie beyond the scope of this section.

Beyond schema-related errors, additional issues arise from a lack of understanding of database structures and the data. The remaining four error types in Figure 11 are handled by *DBTool*, which, when applied to incorrect SQL queries from the baseline, effectively resolved errors related to column relationships, foreign key constraints, and null values (see Section 4). A detailed analysis of whether other methods successfully handled these errors is in Appendix C. Although *DBTool* successfully eliminated execution errors, other types of errors persisted when the baseline SQL selected different columns or tables than the gold SQL—an issue that falls outside the current tool's refinement scope and is deferred to future work.

For both *view* and *DBTool*, many unresolved errors were deemed unsolvable due to the following factors: (1) ambiguity in natural language questions (Floratou et al., 2024), which allows multiple SQL queries as valid answers beyond the gold SQL; (2) errors in the benchmark, including incorrect gold SQL (Wretblad et al., 2024).

6 Conclusion

Database-Centric NL2SQL addresses key limitations of existing NL2SQL frameworks by integrating advanced database functionalities. Through View-based Optimization and Database-as-a-Tool, it simplifies schema handling, enhances query refinement, and reduces inference costs. Experimental results confirm its effectiveness, making Database-Centric NL2SQL a scalable and practical solution for real-world applications.

Limitations

Current evaluations are often limited to specific databases, potentially overlooking challenges present in broader or more dynamic use cases. Future research should explore more generalized and realistic benchmarks to ensure wider applicability of the approach.

View. A key limitation of the current approach is that generating Renamed Views requires manual modifications by users, which can be time-consuming and dependent on human intuition. Future research will explore automated techniques to generate Renamed Views without user intervention, reducing the burden on users and ensuring consistency. Similarly, Customized Views must be predefined, limiting their adaptability to diverse queries. To address this, future work will focus on developing automated mechanisms that analyze user queries, cache frequently used patterns, and dynamically construct Customized Views, thereby improving flexibility and efficiency. Furthermore, the effectiveness of views is inherently tied to how they are defined. Poorly designed views may hinder query performance, highlighting the need for systematic strategies to optimize view definitions. In addition, improving view selection techniques is crucial, as selecting the most appropriate view for a given query directly impacts SQL generation accuracy. Addressing these challenges remains an important avenue for future research.

Database-as-a-Tool. In the benchmark, the number of tables and columns is relatively small, resulting in short query execution times for Database-as-a-Tool. In real-world scenarios with large database schemas, optimizing the tool's algorithm is necessary to improve execution speed. Furthermore, the current Tool-augmented NL2SQL framework employs rule-based logic to invoke tools under specific database or query conditions. A key research direction is to enable the LLM to autonomously invoke necessary tools, transitioning toward a more generalized tool-augmentation approach.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Maciej Brencz. 2024. sql_metadata. Accessed: 2025-02-08.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Hasan Alp Caferoğlu and Özgür Ulusoy. 2024. E-sql: Direct schema linking via question enrichment in text-to-sql. *arXiv preprint arXiv:2409.16751*.

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, and Wei Chen. 2024. Rsl-sql: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073*.

Jipeng Cen, Jiaxin Liu, Zhixu Li, and Jingjing Wang. 2024. Sqlfixagent: Towards semantic-accurate sql generation via multi-agent collaboration. *arXiv preprint arXiv:2406.13408*.

Chroma-core. 2025. Chroma-core.

Xuemei Dong, C. 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.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex Van Grootel, Brandon Chow, Kai Deng, Katherine Lin, Marcos Campos, K. Venkatesh Emani, Vivek Pandit, Victor Shnayder, Wenjing Wang, and Carlo Curino. 2024. Nl2sql is a solved problem... not! In *Conference on Innovative Data Systems Research*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024a. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, et al. 2024b. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv preprint arXiv:2411.08599*.

Parker Glenn, Parag Pravin Dakle, and Preethi Raghavan. 2023. Correcting semantic parses with natural language through dynamic schema encoding. *Preprint*, arXiv:2305.19974.

Alon Y Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal*, 10:270–294.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems*, 36:45870–45894.

CY Hsieh, SA Chen, CL Li, Y Fujii, A Ratner, CY Lee, R Krishna, and T Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2024. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *arXiv preprint arXiv:2405.07467*.

Wenqiang Lei, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the role of schema linking in text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6943–6954, Online. Association for Computational Linguistics.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024a. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3).

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

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Xinyuan Lu, Liangming Pan, Yubo Ma, Preslav Nakov, and Min-Yen Kan. 2024. Tart: An open-source tool-augmented framework for explainable table-based reasoning. *arXiv preprint arXiv:2409.11724*.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*.

Toby Mao. 2023. sqlglot. Accessed: 2025-02-08.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. 2023. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. 2023. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies. *arXiv preprint arXiv:2305.12586*.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*.

PlanetScale. 2024. Operating without foreign key constraints. Accessed: February 13, 2025.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2023. Making language models better tool learners with execution feedback. *arXiv preprint arXiv:2305.13068*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. 2024. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40.

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. *arXiv preprint arXiv:2403.20014*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23.

SQLite. 2024. Sqlite foreign key support. Accessed: 2025-02-08.

Chang-You Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring chain-of-thought style prompting for text-to-sql. *arXiv preprint arXiv:2305.14215*.

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*.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. Mac-sql: A multi-agent collaborative framework for text-to-sql. *Preprint*, arXiv:2312.11242.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Niklas Wretblad, Fredrik Gordh Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. 2024. Understanding the effects of noise in text-to-sql: An examination of the bird-bench benchmark. *arXiv preprint arXiv:2402.12243*.

Yuanzhen Xie, Xinzhou Jin, Tao Xie, MingXiong Lin, Liang Chen, Chenyun Yu, Lei Cheng, ChengXiang Zhuo, Bo Hu, and Zang Li. 2024. Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm. *Preprint*, arXiv:2402.10671.

Beichen Zhang, Kun Zhou, Xilin Wei, Xin Zhao, Jing Sha, Shijin Wang, and Ji-Rong Wen. 2024a. Evaluating and improving tool-augmented computation-intensive math reasoning. *Advances in Neural Information Processing Systems*, 36.

Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024b. Finsql: Model-agnostic llms-based text-to-sql framework for financial analysis. SIGMOD/PODS '24, page 93–105, New York, NY, USA. Association for Computing Machinery.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023a. ACT-SQL: In-context learning for text-to-SQL with automatically-generated chain-of-thought. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3501–3532, Singapore. Association for Computational Linguistics.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023b. Toolcoder: Teach code generation models to use api search tools. *arXiv preprint arXiv:2305.04032*.

Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large language model enhanced text-to-sql generation: A survey. *Preprint*, arXiv:2410.06011.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36:50117–50143.

A Baseline

The following examples illustrate the prompts used in our experiment for the *baseline* and the baseline with description (*baseline w/ desc*). The *baseline* prompt represents the schema in the format **Table(Column1, Column2, ...)**. In contrast, the *baseline w/ desc*. prompt explicitly denotes the relationship between tables and columns using the **Table.Column** format and provides detailed descriptions for each column, including its meaning, possible values (with examples), and data type.

```
### Complete sqlite SQL query only and with no explanation.
## Remember the following caution and do not make same mistakes:
# ┌──────────────────────────────────────────────────────────────┐
# |                    TIPS FOR TEXT-TO-SQL                      |
# └──────────────────────────────────────────────────────────────┘
#
# [Schema]
# frpm(CDSCode, Academic Year, County Code, District Code, …)
# satscores(cds, rtype, sname, dname, cname, enroll12,…)
# schools(CDSCode, NCESDist, NCESSchool, StatusType, County, …)
#
# [Foreign Keys]
# frpm.CDSCode = schools.CDSCode
# satscores.cds = schools.CDSCode#
#
### Question: What is the Percent (%) Eligible Free (K-12) in the
school administered by an administrator whose first name is Alusine.
List the district code of the school.
### Knowledge Evidence: Percent (%) Eligible Free (K-12) = `Free
Meal Count (K-12)` / `Enrollment (K-12)` * 100%
### SQL:'''

### SQL '''
```

Figure 12: Example of *baseline* prompt.

```
### Complete sqlite SQL query only and with no explanation.
## Remember the following caution and do not make same mistakes:
# ┌──────────────────────────────────────────────────────────────┐
# |                    TIPS FOR TEXT-TO-SQL                      |
# └──────────────────────────────────────────────────────────────┘
#
# [Schema]
# frpm.CDSCode:  #The CDSCode column in the free and reduced-price
meals table represents the unique identifier for each school in
California.
# frpm.Academic Year:  #The Academic Year column in the free and
reduced-price meals table indicates the academic year for which the
data is reported. This column is of integer type.
# frpm.County Code:  #The County Code column in the free and
reduced-price meals table of the california_schools database
represents the county code of the school district.
# frpm.District Code:  #The District Code column in the free and
reduced-price meals table of the california_schools database
represents the unique identifier for each school district in
California.
# frpm.School Code:  #This column represents the unique identifier
for each school in the dataset. The identifier is an integer value.
# frpm.County Name:  #This column represents the county name of the
school district. The county name is a text field. Example values
include Tuolumne, Nevada, and Alpine.
# frpm.District Name:  #This column represents the name of the
school district that provides free and reduced-price meals. The
values in this column are text. Example values include Oak View
Union Elementary, Elverta Joint Elementary, and Lucerne Valley
Unified.
# frpm.School Name:  #This column is a text column in the table
'free and reduced-price meals' in the database 'california_schools'.
It contains the name of the school. Example values include 'Neal Dow
Elementary', 'Valley Oaks Charter', and 'Pio Pico Elementary'.
…
#
### Question: What is the Percent (%) Eligible Free (K-12) in the
school administered by an administrator whose first name is Alusine.
List the district code of the school.
### Knowledge Evidence: Percent (%) Eligible Free (K-12) = `Free
Meal Count (K-12)` / `Enrollment (K-12)` * 100%
### SQL:'''

### SQL '''
```

Figure 13: Example of *baseline with desc* prompt.

B View

B.1 Pseudo-Code for View to Base Table.

Algorithm 1 Pseudo-Code for View to Base Table.

Input: SQL query SQL
Output: Transformed SQL query SQL'
function CONVERT_RENAMED_TO_BASE(SQL, r_t_b)
    for all $(r\_col, o\_col) \in r\_t\_o$ **do**
        $SQL \leftarrow$ Replace(SQL, r_col, o_col)
    end for
    return SQL
end function
function CONVERT_VIEW_TO_BASE(SQL, v_t_b)
    for all $(view, map) \in v\_t\_b$ **do**
        if $view \in SQL$ **then**
            base_table $\leftarrow$ map["base_table"]
            for all $(v\_col, b\_col) \in map[\text{"}column\_mapping\text{"}]$ **do**
                $SQL \leftarrow$ Replace(SQL, v_col, b_col)
            end for
            $SQL \leftarrow$ Replace(SQL, view, base_table)
        end if
    end for
    return SQL
end function
if renamed_view **then**
    $SQL' \leftarrow$ convert_renamed_to_base(SQL, r_t_b)
end if
if customized_view **or** combined_view **then**
    $SQL' \leftarrow$ convert_view_to_base(SQL', v_t_b)
end if
return SQL'

The view-to-base table algorithm consists of two functions: 'convert_renamed_to_base' and 'convert_view_to_base', which transform SQL queries by replacing renamed columns and views with their corresponding base table representations. The 'convert_renamed_to_base' function iterates through a mapping of renamed columns to their original names and updates the SQL query accordingly. Likewise, the 'convert_view_to_base' function identifies views referenced in the SQL query, retrieves their associated base tables and column mappings, and replaces them to ensure correct execution.

B.2 Example of Renamed View

Table 5 provides an example of query generation using renamed views in the California Schools database. The query aims to retrieve the district code for schools in Fresno that do not offer a magnet program. In the Gold SQL, the query correctly joins the 'frpm' and 'schools' tables using 'CDSCode' to obtain the 'District Code'. The Baseline SQL, however, directly selects 'District' from 'schools' without performing the necessary join, leading to an incorrect query. The View-Generated SQL addresses this issue by referencing 'frpm'.'District Code' and applying the correct join condition between 'frpm' and 'schools'. This ensures proper schema alignment and accurate query generation.

Q: What is the district code for the School that does not offer a magnet program in the city of Fresco? Rename 'District' -> 'District Name'
Gold SQL: `SELECT T1.'District Code' FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode WHERE City = 'Fresno' AND T2.Magnet = 0;`
Baseline SQL: `SELECT 'District' FROM schools WHERE 'City' = 'Fresno' AND T2.'Magnet' = 0;`
View-Generated SQL: `SELECT 'frpm'.'District Code' FROM frpm JOIN 'schools' ON frpm.CDSCode = schools.CDSCode WHERE 'schools'.'Magnet' = 0 AND 'schools'.'City' = 'Fresno';`

Table 5: Renamed View.

B.3 Unified View

```
### Complete sqlite SQL query only and with no explanation.
## Remember the following caution and do not make same mistakes:
# ┌──────────────────────────────────────────────────────────────┐
# |                    TIPS FOR TEXT-TO-SQL                      |
# └──────────────────────────────────────────────────────────────┘
#
# [Columns]
# CDSCode
# Academic Year
# County Code
# District Code
# cds
# region_type
# school_name
# district_name
# county_name
# …
#
#
### Question: What is the Percent (%) Eligible Free (K-12) in the
school administered by an administrator whose first name is Alusine.
List the district code of the school.
### Knowledge Evidence: Percent (%) Eligible Free (K-12) = `Free
Meal Count (K-12)` / `Enrollment (K-12)` * 100%
### SQL:'''

### SQL '''
```

Figure 14: Example prompt of Unified View.

Figure 14 presents an example of a prompt utilizing the Unified View. Since the Unified View provides a fully joined representation of all tables, only column names are included in the prompt instead of the conventional **table(column1, column2, ...)** format. Notably, these column names are identical to those in the Renamed View.

As shown in Table 6, the Baseline SQL selects 'Enrollment (K-12)', which does not exist in the 'schools' table, leading to an execution error. In contrast, queries generated using the Unified View are algorithmically decomposed into their original

Q: When did the first-through-twelfth-grade school with the largest enrollment open?

Gold SQL:
```
SELECT T2.OpenDate FROM frpm AS T1
ORDER BY T1.Enrollment (K-12) DESC LIMIT;
```

Baseline SQL:
```
SELECT OpenDate FROM schools
ORDER BY 'Enrollment (K-12)' ASC DESC 1;
```

View-Generated SQL:
```
SELECT schools.OpenDate FROM schools
JOIN frpm ON frpm.CDSCode = schools.CDSCode
WHERE frpm.Enrollment (K-12) = (SELECT
MAX(frpm.Enrollment (K-12)) FROM frpm JOIN
schools ON frpm.CDSCode = schools.CDSCode ) AND
schools.OpenDate IS NOT NULL;
```

Table 6: Example of Unified View.

Base tables (Appendix B.1), ensuring the correct mapping to 'frpm.Enrollment (K-12)'.

B.4 Customized View

```
### Complete sqlite SQL query only and with no explanation.
## Remember the following caution and do not make same mistakes:
# ┌──────────────────────────────────────────────────────────────────┐
# |                    TIPS FOR TEXT-TO-SQL                          |
# └──────────────────────────────────────────────────────────────────┘
#
#
# [Schema]
# Examination ('Patient ID', 'Examination Date', 'anto-Cardiolopin antibody (aCL IgG)', …. )
# Patient ( 'Patient ID', 'Patient Sex', 'Patient Birthday', 'Patient Table Record Desc', …)
# Laboratory ( 'Patient ID', ''Laboratory test date (YYMMDD)', AST glutamin oxaloacetic (GOT), …)
#
# [VIEW DDL]
# CREATE VIEW patients_per_month AS
# SELECT CAST(COUNT(T1.ID) AS REAL) / 12 FROM Patient AS T1
# INNER JOIN Laboratory AS T2 ON T1.ID = T2.ID
# ...
#
### Question: For the year that concluded on December 31, 1998, how many male patients on average
were tested in the lab each month?
### Knowledge Evidence: External Knowledge (If Benchmark Provides)
### SQL:""
```

Figure 15: Example prompt of Customized View.

Figure 15 illustrates how a Customized View is incorporated into a prompt, providing a structured representation of the database schema and predefined views. Table 7 demonstrates the impact of utilizing this prompt, showing how defining a view simplifies SQL generation while maintaining correct execution. By leveraging the Customized View 'PatientsPerMonth', the resulting SQL query becomes more concise, eliminating redundant computations and improving interpretability.

B.5 View Selection

Figure 16 illustrates the prompt used in view selection to determine whether an aggregation function or formula is necessary for processing the user question. The prompt guides the LLM to articulate its reasoning using CoT. If the 'Require Aggregation or Formula' returns 'Yes', a Customized View is embedded in the prompt; otherwise, a Renamed View is applied.

Q: For the year that concluded on December 31, 1998, how many male patients on average were tested in the lab each month?

Baseline SQL (Incorrect):
```
SELECT CAST(COUNT(DISTINCT Laboratory.ID) AS
REAL) / 12 AS AvgMalePatientsPerMonth FROM
Laboratory JOIN Patient ON Laboratory.ID
= Pation.ID WHERE Patient.Sec = 'M' AND
Laboratory.Date BETWEEN '1998-01-01' AND
'1998-12-31';
```

View Definition:
```
CREATE VIEW PatientsPerMonth AS SELECT.
CAST(COUNT(T1.ID) AS REAL) / 12 FROM Patient
AS T1 INNER JOIN Laboratory AS T2 ON T1.ID =
T2.ID
```

View-Generated SQL (Correct):
```
SELECT * FROM PatientsPerMonth WHERE
STRFTIME('%Y', T2.Date) = '1998' AND T1.SEX =
'M';
```

Table 7: Example of Customized View.

```
### Determine whether the given user question requires an aggregation function or a
formula. Return the # result in JSON format.
#
# Example Output:
# {
#   "Requires Aggregation or Formula": "Yes or No",
#   "CoT Reasoning": "Step-by-step reasoning explaining the decision"
# }
#
### Question: User Question
### Knowledge Evidence: External Knowledge (If Benchmark Provides)
### Return only the json object without any explanation:
### JSON object:
```

Figure 16: Prompt template of view selection.

C Database-as-a-Tool (DBTool)

C.1 Effect of DBTool on Error Resolution

By analyzing the SQL queries generated by GPT-4o-mini on the California Schools database from BIRD, we identified and categorized 11 errors stemming from a lack of understanding of the database structure and its data. According to Table 8, DBTool proved to be particularly more effective than other existing methods in addressing most of these errors, with the exception of column-value mismatch errors. Additionally, as shown in Table 4 in Section 5, DBTool not only effectively addresses these errors but also demonstrates high token efficiency in solving them.

C.2 Pseudo-Code for DBTool

The simplified algorithm for DBTool is presented in Algorithm 2. DBTool is invoked when specific conditions are met—either in the target database or in the initial SQL query requiring refinement.

Error type	fk_violation			column-value mismatch					null		col-rel
Q_id	10	42	51	18	73	75	76	86	22	43	30
DBTool	O	O	O	O	O	X	X	O	O	O	O
CHESS	O	O	X	X	O	O	O	X	X	X	X
RSL-SL	X	O	X	O	X	X	O	X	X	X	X
E-SQL	X	O	X	O	O	O	O	X	X	X	X

Table 8: Comparison of error resolution across different methods using GPT-4o-mini. Each column corresponds to a specific question ID (Q_id) and error type: FK constraints violation (fk_violation), column-value mismatch, null, and column relationship (col-rel) errors. "O" indicates that the error was rectified, while "X" indicates failure.

Algorithm 2 Pseudo-Code for DBTool

Input: previous SQL SQL, Database D
Output: refined SQL SQL'
result ← Execute SQL on D
conditions ← {
"exec_err": result is error message
"column-value mismatch": result is empty,
"fk_violation": D has FK constraints violation,
"col_rel": Column in D has 1:1 **or** 1:N relationships,
"null": result contains NULL values **or** SQL has arithmetic operations **or** COUNT
}
for all (issue, condition) in conditions **do**
  if condition is True **then**
    SQL' ← DBTool(SQL, issue)
  end if
end for
return refined SQL SQL'

Furthermore, the information retrieved by DBTool from the database varies depending on these conditions. Based on the specific scenario, different predefined SQL templates, user-defined functions for metadata conversion, and guidelines embedded in the LLM prompt for generating refined SQL are employed.

C.3 Example of DBTool-refined Case.

In the remainder of Appendix C, we explore each component of DBTool through an actual example of successful refinement. The example demonstrates how metadata indicating FK violations in the database is leveraged to correct errors in an initial SQL query. Other types of metadata are extracted from the database and incorporated into the LLM's prompt through a similar process.

In Table 9, the question asks for the type of education offered ('EdOpsName') at the school ('cds', 'CDSCode') with the highest average math score ('AvgScrMath'). The SQL generated by the Baseline approach first attempts to find the school with

Q: What is the type of education offered in the school who scored the highest average in Math?

Gold SQL:
```
SELECT T2.EdOpsName FROM satscores AS T1
INNER JOIN schools AS T2 ON T1.cds = T2.CDSCode
ORDER BY T1.AvgScrMath DESC LIMIT 1
```

Baseline SQL:
```
SELECT s.EdOpsName FROM satscores sc
JOIN schools s ON sc.cds = s.CDSCode
WHERE sc.AvgScrMath = (SELECT MAX(AvgScrMath)
FROM satscores);
```

DBTool-refined SQL:
```
SELECT s.EdOpsName FROM satscores sc
JOIN schools s ON sc.cds = s.CDSCode
WHERE sc.AvgScrMath = (SELECT MAX(AvgScrMath)
FROM satscores WHERE cds IN (SELECT CDSCode
FROM schools));
```

Table 9: Example of query refinement using DB-as-a-Tool. The tool corrects the view-generated SQL query by detecting FK constraint violations.

the highest 'AvgScrMath' in the 'satscores' table and then retrieves information about that school by joining it with the 'schools' table. However, the execution returns an empty set. DBTool identifies that this issue arises due to FK constraint violations between the 'satscores' and 'schools' tables.

C.4 SQL Query Parser

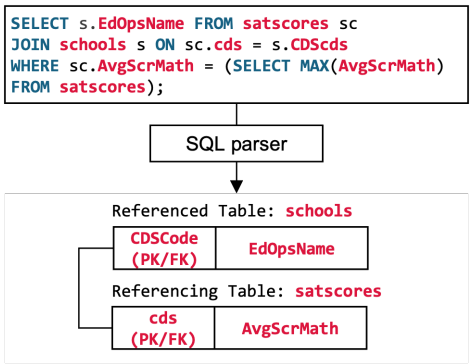


Figure 17: Parsing initial query with SQL parser.

First, column and table names from the previous query are extracted using an SQL parser 17. In the current version of DBTool, the Python libraries 'sqlglot' (Mao, 2023) and 'sql-metadata' (Brencz, 2024) are both used as parsers. For the example above, 'schools' and 'satscores', along with their columns in the query, are extracted by the parser.

C.5 Predefined SQL Template

After parsing the query, the extracted table and column names are inserted into predefined SQL

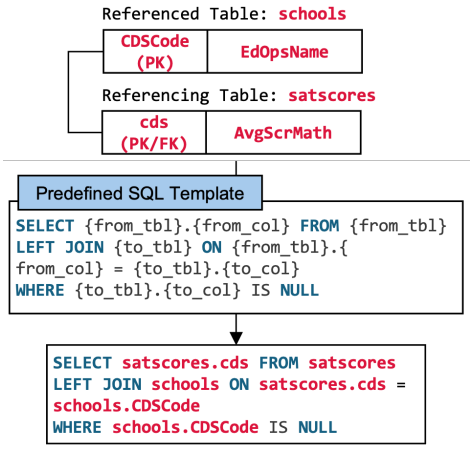


Figure 18: Parameterizing predefined SQL template with parsed tables and columns.

templates (Figure 18). In this example, the SQL template for detecting FK constraints is parameterized with the table 'schools' and its PK 'CDSCode', as well as the table 'satscores' along with its respective FK columns 'cds'.

C.6 User-defined Function (UDF) for Converting Execution Results into Metadata

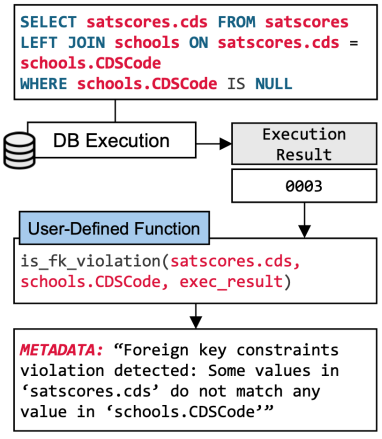


Figure 19: UDF to convert SQL result to metadata.

When parameterized SQL templates from the previous step are executed, their results are processed through the corresponding user-defined function (UDF) that converts them into natural language for better LLM interpretation (Figure 19). In this example, the UDF `is_fk_violation` (Figure 8) takes the execution results, along with the table and column names, as input. If the query returns one or more rows, the function generates a message indicating the presence of an FK constraints violation.

C.7 Chain-of-Thought Reasoning

```
### SCHEMA
{REDUCED DATABASE SCHEMA}

### METADATA
{METADATA}

### OUTPUT
# Question: {QUESTION}
# Knowledge Evidence: {HINT}
# Predicted SQL: {PREVIOUS SQL}
# Execution Result: {EXECUTION RESULT}

### GUIDELINE
{METADATA SPECIFIC GUIDELINES}

### FORMATTING
Your output should be of the following JSON format with no explanation:
{
    "chain_of_thought_reasoning": <str: Your thought process on how you
arrived at the full SQL query.>,
    "refined_sql_query": <str: the full SQL query>
}
```

Figure 20: Prompt template for DBTool

> The predicted SQL attempts to find the type of education offered by the school with the highest average Math score by joining the 'satscores' and 'schools' tables. However, due to the [CONSTRAINT ERROR], some 'cds' values in the 'satscores' table may not have corresponding entries in the 'schools' table. This means that a direct join might exclude the highest scoring entry if it lacks a corresponding school entry. To address this, we should first find the maximum average Math score from the 'satscores' table, then join the 'satscores' and 'schools' tables using this maximum score, ensuring that we only consider entries that have a corresponding school. The subquery should be used to find the maximum score, and then the main query should join the tables based on this score, ensuring that the FK constraint error is considered.

Figure 21: Example of a CoT reasoning (GPT-4o).

When the metadata is returned from the UDF, it is included in the prompt for generating a refined SQL query (Figure 20). The prompt utilizes the CoT technique (Wei et al., 2022), widely used in NL2SQL task for SQL generation (Talaei et al., 2024), (Caferoğlu and Ulusoy, 2024), and (Lee et al., 2024).

According to the actual CoT reasoning (Figure 21), GPT-4o effectively incorporates the provided metadata and refines the subquery to ensure that the highest 'AvgScrMath' is selected only from schools present in both the 'satscores' and 'schools' tables. Finally, initial SQL is refined as the correct answer, producing an execution result that matches the output of the Gold SQL.

C.8 Predefined SQL Templates for Other Types of Metadata

Figure 22 and Figure 23 are the predefined SQL templates used by DBTool to check column relationships and detect the presence of null values in the database, respectively.

```
SELECT COUNT(*)
FROM (
    SELECT `{colA}`, COUNT(DISTINCT `{colB}`) AS
          colB_count FROM {table}
    WHERE `{colA}` IS NOT NULL AND `{colB}` IS
         NOT NULL
    GROUP BY `{colA}` HAVING colB_count > 1
);
```

Figure 22: SQL to check the column relationships.

```
SELECT COUNT(*) FROM {tbl}
WHERE `{col}` IS NULL;
```

Figure 23: SQL to check the presence of null values.

C.9 User-defined Function (UDF) for Other Types of Metadata

```
def is_null(tbl, colA, colB, exec_result):
    exec_result[0], exec_result[1] = colB_cnt,
         colA_cnt
    if colB_cnt > 0 and colA_cnt > 0:
        # colA : colB = N : M (not useful)
        return None
    elif colB_cnt > 0:
        # colA : colB = 1 : N
        return(f"There are multiple {colB} for
            each {colA}")
    elif colA_cnt > 0:
        # colA : colB = M : 1
        return(f"There are multiple {colB} for
            each {colA}")
    else:
        # colA : colB = 1 : 1
        return(f"Each {colB} has one
            corresponding {colB}")
```

Figure 24: UDF to check the column relationships.

Figure 24 and Figure 25 are the simplified codes of user-defined function that converts an execution result into metadata for the column relationships and the presence of null values, respectively.

C.10 Handling Other Common NL2SQL Errors with DBTool

As discussed in Section 4, DBTool resolves common NL2SQL errors, including column-value mismatches and execution errors, using a different approach from SQL query refinement based on metadata extraction (FK constraint violations, column relationships, and null values).

To handle column-value mismatches, DBTool extracts column-value pairs from the WHERE clause and verifies their existence in the database. It then identifies problematic column-value pairs

```
def is_null(tbl, col_lst, exec_result):
    null_col = []
    for col in col_lst:
        if exec_result > 0: # If the result is
            more than 0, null value exists in
            the column
            null_col.append(col)
    return (f"Following columns in table {tbl}
         have null values:\n {null_col}")
```

Figure 25: UDF to check the null value presence.

and retrieves all unique values from the database for the corresponding column. Next, DBTool performs a syntactic match between the problematic value and the retrieved unique values, selecting the top five most similar values. These similar values, along with the column name, are provided to the LLM to indicate which values are valid in the column, enabling the LLM to correctly match values in the WHERE clause.

For execution errors, DBTool executes the initial SQL query and extracts problematic column or table names from the error message. If they are missing, the raw message and predefined guidelines for different error types are passed to the LLM. If a column or table does not exist in the schema, DBTool provides the complete schema to help the LLM determine the correct name. Since this process focuses solely on name correction, additional descriptions are unnecessary. If a column exists but is linked to the wrong table, DBTool retrieves its actual table and guides the LLM to correct the mismatch. Notably, even after execution errors are resolved and queries run successfully in the database, other types of errors may still remain. Therefore, DBTool further refines these errors in its algorithm.

In both processes, DBTool utilizes an SQL query parser and database schema or value lookups, enabling efficient error refinement with minimal token usage.

D Performance Comparison by Database

Table 10 summarizes the execution accuracy of different configurations by databases in the BIRD benchmark, categorized by the LLM used. While the performance of *View+DBTool* varies across databases, it improves over *Baseline w/ desc.* by at least 4% and up to 35% for GPT-4o, except for the Toxicology DB.

Database	Baseline	Baseline w/ desc.	View	DBTool	View + DBTool
California Schools	41.57	48.31	48.31	51.69	61.79
Student Club	72.78	67.09	79.75	75.95	84.81
Superhero	79.07	85.27	84.50	83.72	89.15
Debit Card Specializing	50.00	51.56	60.94	54.69	64.06
European Football	66.67	68.99	72.09	71.32	76.74
Formula 1	39.66	48.85	54.60	48.85	58.05
Codebase Community	66.67	66.67	73.12	69.35	74.19
Thrombosis Prediction	52.76	47.24	56.44	55.21	63.80
Financial	51.89	51.89	60.38	55.56	66.04
Toxicology	48.28	68.28	63.45	59.31	65.52
Card Games	53.93	57.07	58.64	61.26	67.54
All	57.30	60.63	65.25	63.03	70.47

(a) Model: GPT-4o

Database	Baseline	Baseline w/ desc.	View	DBTool	View + DBTool
California Schools	33.71	32.58	38.20	46.04	52.81
Student Club	68.99	70.89	70.25	72.78	72.78
Superhero	76.74	81.40	81.40	84.50	86.05
Debit Card Specializing	40.62	48.44	50.00	50.00	56.25
European Football	57.36	58.91	62.79	67.44	68.99
Formula 1	41.95	42.53	42.53	49.43	45.40
Codebase Community	58.06	58.60	61.29	61.83	65.59
Thrombosis Prediction	47.24	44.17	47.85	63.19	63.19
Financial	43.40	44.34	49.06	50.00	53.77
Toxicology	45.52	64.83	55.17	52.41	61.38
Card Games	49.21	43.46	47.64	54.97	53.40
All	52.28	54.24	55.54	60.10	61.93

(b) Model: GPT-4o-mini

Database	Baseline	Baseline w/ desc.	View	DBTool	View + DBTool
California Schools	32.58	3.37	35.96	47.19	51.69
Student Club	65.19	70.89	71.52	74.05	80.38
Superhero	75.97	4.65	73.64	86.82	90.70
Debit Card Specializing	48.44	54.69	39.06	57.81	57.81
European Football	65.89	15.50	64.34	69.77	70.54
Formula 1	40.80	46.55	39.66	54.02	50.00
Codebase Community	62.90	54.84	62.90	72.04	72.58
Thrombosis Prediction	50.92	47.85	50.92	60.12	60.12
Financial	38.68	5.66	39.62	50.94	54.72
Toxicology	21.38	25.52	28.28	40.00	48.97
Card Games	47.64	47.12	46.07	56.69	59.16
All	50.85	37.16	51.37	61.55	63.89

(c) Model: Llama 3.1-70B

Table 10: Execution Accuracy (%) by database in the BIRD benchmark.