RISE: Rule-Driven SQL Dialect Translation via Query Reduction

Xudong Xie*
Institute of Software Chinese Academy of Sciences
Beijing, China
xiexudong23@otcaix.iscas.ac.cn

Yuwei Zhang*‡
Institute of Software Chinese Academy of Sciences
Beijing, China
zhangyuwei@iscas.ac.cn

Wensheng Dou*†‡
Institute of Software Chinese Academy of Sciences
Beijing, China
wsdou@otcaix.iscas.ac.cn

Yu Gao*
Institute of Software Chinese Academy of Sciences
Beijing, China
gaoyu15@otcaix.iscas.ac.cn

Ziyu Cui*
Institute of Software Chinese Academy of Sciences
Beijing, China
cuiziyu20@otcaix.iscas.ac.cn

Jiansen Song*
Institute of Software Chinese Academy of Sciences
Beijing, China
songjiansen20@otcaix.iscas.ac.cn

Rui Yang*
Institute of Software Chinese Academy of Sciences
Beijing, China
yangrui22@otcaix.iscas.ac.cn

Jun Wei*†
Institute of Software Chinese Academy of Sciences
Beijing, China
wj@otcaix.iscas.ac.cn

Abstract

Translating SQL dialects across different relational database management systems (RDBMSs) is crucial for migrating RDBMS-based applications to the cloud. Traditional SQL dialect translation tools rely on manually-crafted rules, necessitating significant manual effort to support new RDBMSs and dialects. Although large language models (LLMs) can assist in translating SQL dialects, they often struggle with lengthy and complex SQL queries.

In this paper, we propose RISE, a novel LLM-based SQL dialect translation approach that can accurately handle lengthy and complex SQL queries. Given a complex source query Q_c that contains a SQL dialect d, we first employ a dialect-aware query reduction technique to derive a simplified query Q_s by removing d-irrelevant SQL elements from Q_c. Subsequently, we utilize LLMs to translate Q_s into $Q_{s'}$, and automatically extract the translation rule r_d for dialect d based on the relationship between Q_s and $Q_{s'}$. By applying r_d to Q_c, we can effectively translate the dialect d within Q_c, thereby bypassing the complexity of the source query Q_c. We evaluate RISE on two real-world benchmarks, i.e., TPC-DS and SQLProcBench, comparing its performance against both the traditional rule-based tools and the LLM-based approaches with respect to translation accuracy. RISE achieves accuracies of 97.98% on TPC-DS and 100% on SQLProcBench, outperforming the baselines by an average improvement of 24.62% and 238.41%, respectively.

CCS Concepts

• **Software and its engineering** → **Software maintenance tools**;
• **Computing methodologies** → *Machine translation.*

Keywords

Large language model, SQL dialect, Database management system

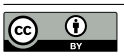
ACM Reference Format:
Xudong Xie, Yuwei Zhang, Wensheng Dou, Yu Gao, Ziyu Cui, Jiansen Song, Rui Yang, and Jun Wei. 2026. RISE: Rule-Driven SQL Dialect Translation via Query Reduction. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773257

1 INTRODUCTION

Nowadays, more than 50 relational database management systems (RDBMSs) are widely used across organizations [6], many of which have been deployed on the cloud to enhance performance and reduce costs. Different RDBMSs typically offer distinct advantages; for example, some excel in scalability, while others excel in security. To achieve these benefits, legacy RDBMS-based applications are expected to migrate into suitable cloud RDBMSs [30, 38, 42].

Although many RDBMSs (e.g., Oracle [14], MySQL [12], PostgreSQL [15] and SQLite [17]) support Structured Query Language (SQL), they incorporate vendor-specific extensions and unique nuances, resulting in a wide variety of SQL dialects. The presence of these diverse SQL dialects introduces both syntax and semantic discrepancies, making SQL dialect translation across different RDBMSs a challenging problem. Effective SQL dialect translation should ensure semantic equivalence before and after translation, meaning that the same query results and database states should

*Affiliated with Key Lab of System Software at CAS, State Key Lab of Computer Science at Institute of Software at CAS, and University of CAS, Beijing. CAS is the abbreviation of Chinese Academy of Sciences.
†Affiliated with Nanjing Institute of Software Technology, University of Chinese Academy of Sciences, Nanjing.
‡Yuwei Zhang and Wensheng Dou are the corresponding authors.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773257

be returned. Existing study [9] has demonstrated that SQL dialect translation typically consumes 20-40% of migration budgets.

Several rule-based tools [5, 11, 16, 17] have been developed to support SQL dialect translation across RDBMSs. The widespread adoption of tools (e.g., SQLGlot [16] with 7.8k GitHub stars) and proprietary solutions from Microsoft and Amazon [1, 4] highlight the critical need for addressing SQL dialect translation challenges in both academia and industry [42]. These tools mainly rely on manually-crafted translation rules, which are often incomplete and error-prone. For example, the complex query Q_c shown in Figure 1 cannot be correctly translated by existing tools. Specifically, jOOQ and SQLines do not support the `FULL OUTER JOIN` clause, while SQLGlot generates an incorrect query by substituting `UNION ALL` for `UNION`, resulting in inconsistent query outcomes. Moreover, the manual effort required to develop and maintain these translation rules is considerable. For example, SQLGlot includes 1900 lines of code dedicated to MySQL-to-PostgreSQL translation.

Recently, researchers have explored the use of large language models (LLMs) to reduce dependence on manually-crafted rules in traditional tools [5, 11, 16, 17]. For example, Mallet [30] employs LLMs to generate translation rules, which are then manually integrated into SQLGlot [16] by developers. Therefore, Mallet inherits many of the limitations associated with SQLGlot. A more direct approach involves prompting LLMs to translate SQL queries, referred to as `LLMTranslator`, which is an experimental prototype designed in this paper. However, `LLMTranslator` struggles with lengthy and complex queries due to hallucination issues commonly observed in LLMs [22]. For example, when translating the complex query Q_c from Figure 1, `LLMTranslator` incorrectly rewrites '`sum(sum(ws_sales_price))`' as '`sum(ws_sales_price)`', altering the intended semantics of the query.

To assess the practical limitations of `LLMTranslator`, we conduct an empirical study using two benchmarks (i.e., TPC-DS [18] and SQLProcBench [24]), both of which are derived from real-world workloads. The results demonstrate that `LLMTranslator` performs well on simple queries, accurately translating SQL dialects with straightforward syntax and semantics. However, its effectiveness declines markedly when handling lengthy and complex queries. In such cases, `LLMTranslator` frequently exhibits hallucination issues, leading to incorrect translations. These findings suggest that while `LLMTranslator` excels in simple translation tasks, it struggles with complex queries due to the presence of dialect-irrelevant noise that can obscure the essential semantics. Therefore, isolating and filtering out such non-essential elements is crucial for improving the accuracy of LLM-based SQL dialect translation.

Inspired by the above observations, we propose `RISE`, an automated **R**ule-driven SQL d**I**alect tran**S**lation approach that utilizes dialect-aware query r**E**duction. The core idea is to simplify complex SQL queries by removing dialect-irrelevant elements. This reduction technique produces simplified queries that retain only the elements specific to the target SQL dialect. As illustrated in Figure 1, the source query Q_c spans 42 lines, features complex structures, and includes two PostgreSQL-specific dialects (colored in red). Through our reduction technique, Q_c is first transformed into a simplified query Q_s, which isolates the dialect-specific elements of `FULL OUTER JOIN` in two lines. We then prompt LLMs to translate Q_s into $Q_{s'}$, a task that LLMs can handle effectively due to the reduced complexity. From the correspondence between Q_s and $Q_{s'}$, we generate a translation rule r_d for the first dialect `FULL OUTER JOIN`. Finally, we apply r_d back to the original Q_c, enabling accurate and efficient translation of dialect-specific elements without requiring LLMs to process the entire complex query.

We evaluate `RISE` on two real-world benchmarks, i.e., TPC-DS and SQLProcBench, which require SQL dialect translations from PostgreSQL to MySQL and Oracle, respectively. We compare `RISE` against state-of-the-art baselines, including traditional rule-based tools (i.e., SQLines, jOOQ and SQLGlot) and LLM-based methods (i.e., our prototype `LLMTranslator` and CrackSQL). Experimental results show that `RISE` substantially surpasses all baselines in terms of translation accuracy. We have made our tool available at https://figshare.com/s/5c709f5c0a63d58b0ef4.

In summary, we make the following contributions.

- We conduct the first empirical study to assess the effectiveness and limitations of `LLMTranslator`, offering valuable insights for improving LLM-based SQL dialect translation.
- We propose `RISE`, a novel LLM-based SQL dialect approach capable of accurately handling lengthy and complex SQL queries. `RISE` integrates an effective dialect-aware query reduction technique with an LLM-assisted translation rule generator to overcome the limitations of naive LLM translation.
- We perform extensive experiments on two real-world benchmarks and thoroughly evaluate each component of `RISE`. The experimental results demonstrate that `RISE` outperforms state-of-the-art baselines, highlighting its superior performance.

2 EMPIRICAL STUDY

Given the extensive knowledge and strong natural language understanding capabilities of LLMs, leveraging them for SQL dialect translation offers a highly promising and flexible approach. To assess the effectiveness of LLMs in translating SQL dialects, we implement a prototype `LLMTranslator` and conduct an empirical study aimed at addressing the following two research questions.

- **RQ1 (Translation accuracy):** Can `LLMTranslator` accurately translate SQL dialects?
- **RQ2 (Failure factors):** What factors can lead to failures in SQL dialect translation when using `LLMTranslator`?

2.1 Study Methodology

We introduce the benchmarks, methodology, and evaluation metric used in the empirical study below.

2.1.1 Benchmarks. We evaluate the capability of LLMs in SQL dialect translation using queries from the following two benchmarks:

- **TPC-DS:** TPC-DS [18] is an industry-standard benchmark for evaluating the performance of RDBMSs on complex analytical queries. It consists of 99 queries covering a wide range of SQL constructs, with some queries exceeding 100 lines. These queries include basic operators, aggregations, functions, and advanced SQL features (e.g., `WITH` clauses, `CTE`, sub-queries). TPC-DS captures the intricacies and diversity of real-world query workloads, with each query meticulously designed to simulate practical analytical tasks. Thus, we employ TPC-DS to evaluate the ability of LLMs in addressing the multifaceted challenges of industrial-scale SQL dialect translation.

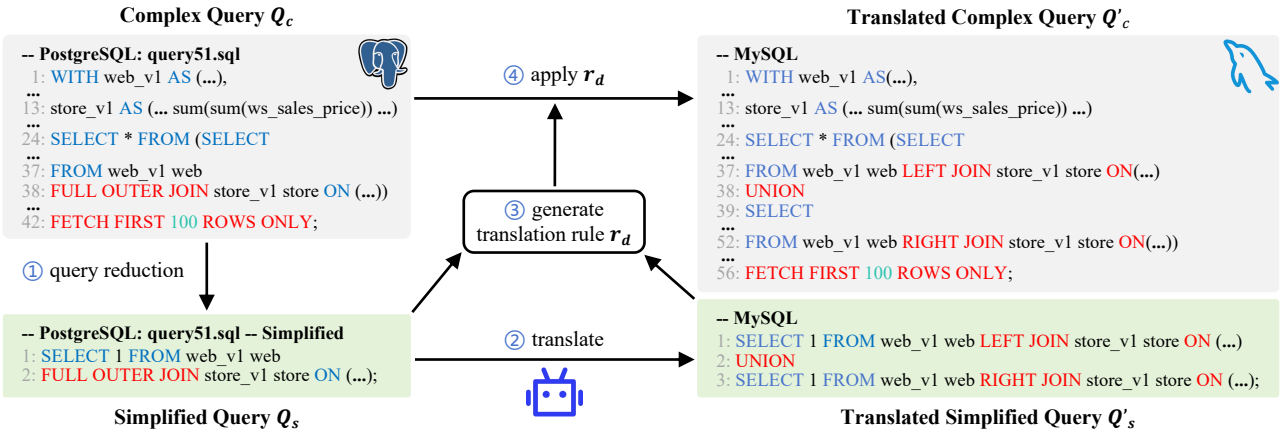


Figure 1: An Example of SQL Dialect Translation. PostgreSQL-Specific Dialects are Colored in Red.

- **SQLProcBench:** SQLProcBench [24] extends the schema of TPC-DS by emphasizing procedural code, including stored procedures, functions, and triggers implemented in PostgreSQL, Oracle, and SQL Server. In our evaluation, we focus on stored procedures, which comprise 44 statements designed to simulate real-world procedural code usage in various applications, with some exceeding 50 lines of SQL code. These procedures exhibit complex control flows (e.g., `IF-ELSE` conditions, `WHILE` and `FOR` loops), individual SQL statements (e.g., `SELECT`, `UPDATE`, `DELETE`), variable management, exception handling, and others. By capturing the inherent challenges of translating procedural logic, SQLProcBench serves as a critical benchmark for evaluating the effectiveness of SQL dialect translation approaches in practical database migration scenarios.

2.1.2 Methodology. We develop a prototype, `LLMTranslator`, to evaluate LLM-based SQL dialect translation and to serve as a baseline for subsequent comparisons. `LLMTranslator` adopts the Chain-of-Thought (CoT) strategy [27], executing the translation process in three stages. First, we execute the source query on both the target and source databases to collect feedback, which serves as error messages (e.g., execution errors or result inconsistencies). Second, we use LLMs to identify and summarize dialect-specific features present in the source query. Finally, LLMs complete the translation by incorporating both the error feedback and the summarized dialect-specific features.

To further assess the contribution of error messages and dialect-specific feature summaries to translation performance, we conduct an ablation study on `LLMTranslator`. Specifically, we evaluate three variants: one excluding error messages (w/o `ErrMsg`), one excluding feature summaries (w/o `Summary`), and one omitting both (`SingleLLM`). To ensure the generalizability of our findings, we evaluate `LLMTranslator` using two representative LLMs: DeepSeek-V3 [8] (denoted as LLMTranslator$_{DS}$) and GPT-4o [10] (denoted as LLMTranslator$_{GPT}$). For each LLM and its variants, we compute the number of syntax errors and inequivalent queries, allowing a maximum of three translation iterations per query.

Table 1: Accuracy of `LLMTranslator` and Its Variants

Model	Syntax error	Inequivalent	Accuracy
`LLMTranslator`$_{GPT}$	7	12	86.71%
w/o `ErrMsg`	7	18 (50.00% ↑)	82.52%
w/o `Summary`	9 (28.57% ↑)	17 (41.67% ↑)	81.82%
`SingleLLM`	13 (85.71% ↑)	21 (75.00% ↑)	76.22%
`LLMTranslator`$_{DS}$	9	16	82.52%
w/o `ErrMsg`	10 (11.11% ↑)	24 (50.00% ↑)	76.22%
w/o `Summary`	12 (33.33% ↑)	20 (25.00% ↑)	77.62%
`SingleLLM`	14 (55.56% ↑)	27 (68.75% ↑)	71.33%

2.1.3 Metric. The accuracy of translations is quantified and used as the evaluation metric. We execute the translated query on the target DBMS and compare its results with those obtained by running the source query on the source DBMS. If the execution fails or produces inconsistent outcomes, the translation is deemed unsuccessful. For queries with consistent results, we further conduct manual verification to confirm semantic equivalence between the translated query and the source query. The translation is considered successful only if semantic consistency is verified.

2.2 Translation Accuracy

As indicated by the empirical results in Table 1, even when using a pure LLM (i.e., `SingleLLM`) directly, most SQL dialects can be successfully translated, with GPT-4o and DeepSeek-V3 achieving accuracy rates of 76.22% and 71.33%, respectively.

> ***O1**: A pure LLM alone can correctly translate the majority of dialects.*

When error messages are available, the LLM can identify the location and the nature of dialect-specific elements, which enhances the ability to accurately translate SQL dialects. Similarly, the feature summaries assist the LLM in achieving a more accurate understanding of the various characteristics within the source query, thereby

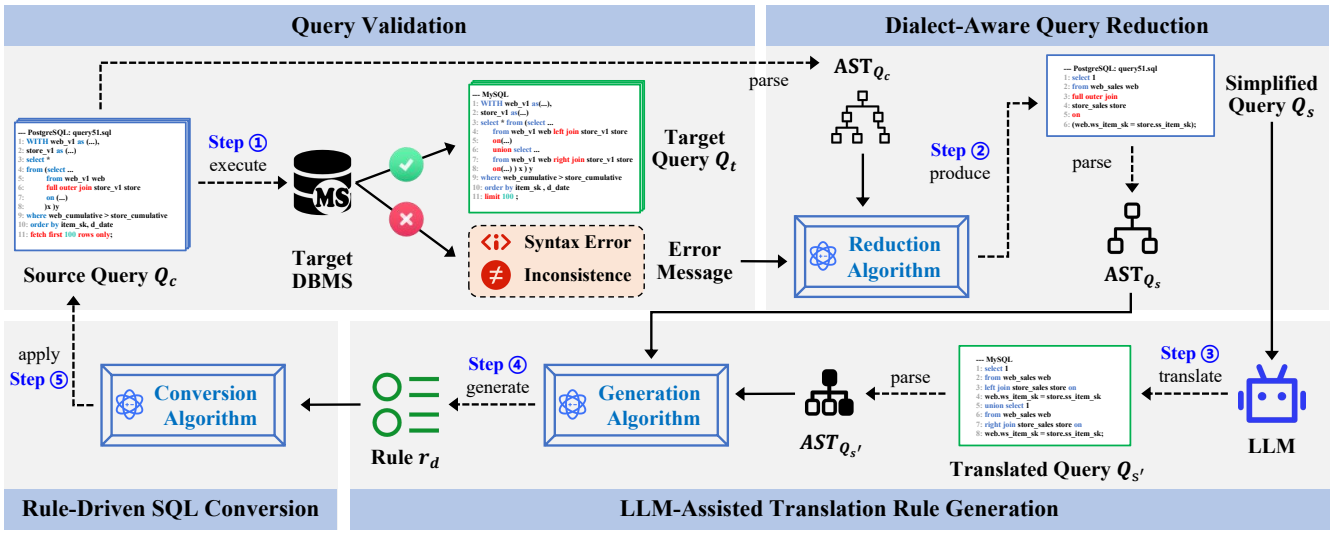


Figure 2: Overview of RISE.

reducing erroneous translations. Removing either the error messages (i.e., w/o `ErrMsg`) or the feature summaries (i.e., w/o `Summary`) increases the number of syntax errors and inequivalent queries by approximately 34.09% and 31.82%, respectively.

> ***O2**: Both error messages and dialect-specific feature summaries can enhance the accuracy of SQL dialect translation.*

2.3 Failure Factors

We conduct a detailed analysis of the erroneous queries produced by $\texttt{LLMTranslator}_{GPT}$ and $\texttt{LLMTranslator}_{DS}$. For these 44 error cases, we first identify the dialect-specific elements. Then, we manually remove dialect-irrelevant content to create simplified versions and re-translate these using `LLMTranslator`.

We find that 38.64% of the errors are directly resolved after removing dialect-irrelevant content, as this prevents inconsistencies caused by the LLM modifying content unrelated to the dialect. For instance, the LLM alters `GROUP BY` clauses (e.g., changing `GROUP BY a, b, c` to `GROUP BY a, b`), modifies column names and constants such as strings within SQL queries, and performs inequivalent optimizations. These include rewriting expressions like `Sum(Sum())` to `Sum()`, as well as converting implicit joins into explicit `JOIN` operations, albeit with incorrect join conditions.

Among the remaining errors, eight were produced by GPT-4o and 19 from DeepSeek-V3. After applying query reduction, $\texttt{LLMTranslator}_{GPT}$ successfully translated all eight previously error cases. In five of these, GPT-4o had initially omitted dialects that required handling. These dialects were correctly processed via query reduction. In the remaining three cases, GPT-4o was misled by verbose information, resulting in incorrect translations of the semantically inequivalent `ROLLUP` function. After reduction, however, GPT-4o accurately identified the subtle semantic differences and produced correct translations. For $\texttt{LLMTranslator}_{DS}$, 11 of the 19 translation failures involved incorrect handling of the `ROLLUP` function, seven were due to unprocessed dialects, and one involved a misinterpretation of the `FULL OUTER JOIN` clause. Following reduction, all queries except those involving `ROLLUP` were successfully translated. These results indicate that `LLMTranslator` can effectively handle most SQL dialects in simplified queries, but struggles with such tasks when dealing with more complex queries. To better understand `LLMTranslator`'s limitations, we further analyze the causes of failure and identify three major types of LLM hallucinations that significantly degrade translation accuracy.

- LLMs make improper optimizations in dialect-irrelevant parts.
- Although LLMs can translate dialects, they often fail to accurately handle complex contexts.
- LLMs arbitrarily modify column names, omit certain elements (e.g., items in the GROUP BY clause), or alter query objects.

> ***O3**: LLMs exhibit limitations in effectively processing lengthy and complex SQL queries.*

3 SQL Dialect Translation via Query Reduction

We propose `RISE`, an automated rule-driven SQL dialect translation approach based on LLMs. Figure 2 shows the overview of `RISE`. Let d be the dialect that needs to be translated. We start by validating the source query Q_c on the target DBMS (Step ①). If Q_c does not produce syntax errors and yields identical results on its target DBMS, it does not require translation and is directly returned as the target query Q_t. Otherwise, we assume that the existence of dialect d. Then, we perform a dialect-aware query reduction on Q_c, in which an adaptive parser converts Q_c into its corresponding abstract syntax tree (AST) AST_{Q_c}. Using the error message and AST_{Q_c}, the query reduction algorithm produces a simplified query Q_s while retaining dialect d (Step ②). In the next step, we directly utilize LLMs to translate Q_s into $Q_{s'}$ (Step ③). An adaptive parser then converts both the simplified and translated queries, i.e., Q_s and $Q_{s'}$, into ASTs (i.e., AST_{Q_s} and $AST_{Q_{s'}}$), and a rule generation algorithm derives the new translation rule r_d of dialect d from these ASTs (Step ④). Finally, we leverage r_d to translate the dialect d

within Q_c (Step ⑤) and subsequently perform query validation on Q_c. If Q_c still contains unresolved dialect elements or the generated rule fails to translate the dialect d in Q_c, we repeat the above process. Otherwise, we return Q_c as the target query Q_t.

3.1 Query Validation

We perform query validation at the following three stages of the process. First, we evaluate the source query Q_c by treating it as a translated query on the target DBMS to detect its SQL dialect and retrieve error messages. Second, we verify the LLM-generated translations to ensure that SQL dialects have been accurately translated. Lastly, we assess rule-driven translations to confirm proper dialect handling and identify unresolved SQL dialects.

We adopt a dynamic execution strategy to determine whether the translated query contains unresolved dialect-specific elements and capture associated error messages. Specifically, we execute both the pre-translation and post-translation queries on their respective source and target DBMSs and then compare their outcomes. Before each validation, we reset the database to its initial state, ensuring consistent evaluation conditions regardless of any intermediate statements that may have been executed. We conduct validation using the table data included in the Benchmarks. If the Benchmark does not contain such data, these data can be generated using existing methods [19, 23, 26]. The presence of unresolved dialects in the translated query is identified under the following conditions:

(1) The target DBMS fails to execute the translated query, and the DBMS feedback serves as the error message.
(2) The target DBMS returns different query results from those returned by the source query, and the discrepancies between the results serve as the error message.
(3) The target DBMS stores different database states after executing the query that is intended to modify the database state.

Using this straightforward yet practical method, similar to the approaches adopted by Mallet [30] and VeriEQL [25], we can effectively verify the syntactic validity and semantic equivalence of the SQL queries before and after translation.

3.2 Dialect-Aware Query Reduction

If dialects are identified during query validation step, we employ an adaptive parser to generate the corresponding abstract syntax tree (AST) and perform subsequent query reduction.

3.2.1 Adaptive Parsing. Inspired by SQLess [28], we implement an adaptive parser using ANTLR [2] to parse SQL queries into their corresponding ASTs. We leverage the g4 grammar file [3] of the source DBMS to generate a basic parser by ANTLR. Then, we utilize the basic parser to parse the queries into ASTs. If a parsing error occurs, we will generate a patch for the original g4 grammar file to complete the parsing rules and resolve the issue. This step is similar to SQLess in achieving an adaptive parser. However, SQLess does not analyze the semantics of unparseable parts. Instead, it directly adds the unparseable strings into the error patterns of the parsing rule, without creating finer-grained parsing rules for the failed strings. In contrast, we use an LLM, guided by the parsing error message, to analyze the unparseable parts and generate new parsing rules. These rules are then manually validated, and the legitimate ones are integrated into the original g4 grammar file. This approach facilitates the construction of semantically richer patches and the development of a more robust and versatile adaptive parser.

3.2.2 Query Reduction. This component focuses on removing as much dialect-irrelevant elements as possible from the source query while preserving only the dialect-specific elements. Note that many program reduction methods [29, 33, 34, 36, 39, 40] have been proposed. In our approach, we leverage SQL's AST structure for syntax-based reduction. Specifically, we adopt a lightweight yet effective method[1] to randomly delete removable subtrees from the query's AST. Subsequently, we employ an LLM to perform further query reduction, refining the remaining query structure.

Step 1: Syntax-based random reduction. We begin by applying random reduction to the AST parsed from the source query. In each iteration, up to two subtrees are randomly selected for removal, resulting in a simplified AST and its corresponding simplified query. The resulting simplified query must satisfy two constraints: (1) It must be successfully executed on the source DBMS; (2) It must still produce the same error message on the target DBMS. The first constraint ensures that no new errors are introduced during reduction, while the second guarantees that the dialect-specific elements are fully preserved. If the simplified query fails to meet both constraints, the removed subtrees are restored, and alternative subtrees are selected for the next iteration. This process continues until no further subtrees can be removed.

Listing 1: Example Prompt for LLM-Based Reduction

```
Task description: Migrate the query from {source_db} to
     {target_db}. Simplify the query below to retain
    only parts needed to trigger '{original_error}' in
     {target_db}.
Query: {sql}
CoT instructions: Locate the error, analyze its cause,
    and remove irrelevant content.
Requirements:
1: Keep the query as short as possible.
2: Using original table/column names.
3: Ensure the simplified query compiles on {source_db}.
4: Preserve the dialect part intact.
Return format:```sql the simplified query here ```
```

Step 2: LLM-based reduction. We then apply an LLM to further refine the query, as random reduction alone may retain dialect-irrelevant elements under complex constraints (e.g., subqueries and CTE). An LLM is used to extract and understand the dialect-specific portions, retaining only the essential parts of the simplified query produced by the random reduction method. The example prompt is illustrated in Listing 1. We avoid using an LLM directly on the source query for reduction, as our empirical observations indicate that LLMs can introduce various errors when processing lengthy SQL queries. In this step, the LLM generates five outputs for each reduction attempt. For each output, we compute a score by executing the resulting query on the target DBMS, retrieving the new error message, and calculating the similarity between the original and new error messages. If none of the outputs exceed a similarity score of 0.85 (an empirically determined threshold), it suggests that the original dialect has not been fully preserved. We collect the failed attempts along with their causes of failure and feed this information back into the next round of the LLM

[1] In our experiments, our random reduction strategy takes an average of 16 seconds to reduce a SQL query, achieving a reduction rate of 91%.

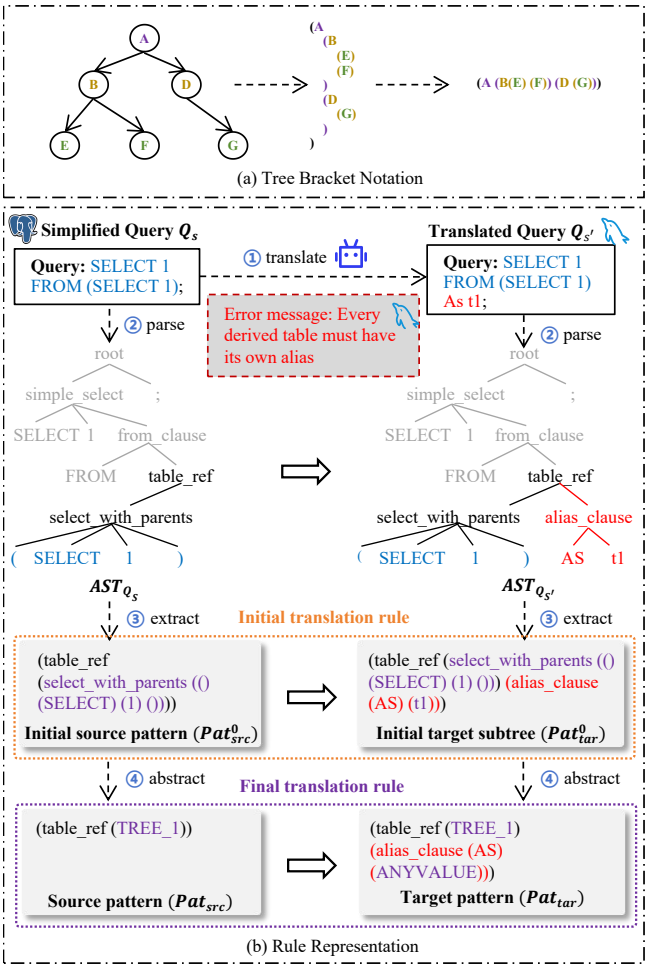


Figure 3: An Illustrative Example of Translation Rule Generation.

prompting. This process repeats iteratively, up to a maximum of five iterations. Once an output meets the criteria, we execute it on the source DBMS. If this execution is error-free, the output is accepted as final simplified query; otherwise, it is discarded.

3.3 LLM-Assisted Translation Rule Generation

This module is divided into two primary components: query translation and rule generation. It utilizes LLMs to accurately translate the simplified query and employs a rule generation algorithm to automatically derive translation rules based on the relationship of both the simplified and translated queries.

3.3.1 Query Translation. After obtaining the simplified query, we utilize `LLMTranslator` described in Section 2.1.2 to perform the SQL dialect translation. The translated query undergoes the validation process outlined in Section 3.1. Any translation that fails to execute or produces results inconsistent with the simplified query is discarded, and `LLMTranslator` is re-prompted with the simplified query to generate new translations.

Algorithm 1: Initial Translation Rule Extraction

Input: Simplified query AST AST_{Q_s} and its corresponding translated query AST $AST_{Q_s'}$

Output: Initial translation rule $\langle Pat^0_{src}, Pat^0_{tar} \rangle$

1 $\langle Pat^0_{src}, Pat^0_{tar} \rangle \leftarrow$ `getMinDiffTree`$(AST_{Q_s}, AST_{Q_s'})$
2 **Function** `getMinDiffTree`(T_{src}, T_{tar})
3 **if** $AST_{Q_s}.remove(T_{src}) \neq AST_{Q_s'}.remove(T_{tar})$ **then**
4 **return** $NULL$
5 **else**
6 $num \leftarrow max(T_{src}.root.childNum, T_{tar}.root.childNum)$
7 **for** $i \leftarrow 0; i < num; i \leftarrow i+1$ **do**
8 $child_1 \leftarrow T_{src}.root.getChild(i)$
9 $child_2 \leftarrow T_{tar}.root.getChild(i)$
10 **if** $child_1 \neq child_2$ **then**
11 **return** $\langle T_{src}, T_{tar} \rangle$
12 **else**
13 $rule \leftarrow$ `getMinDiffTree`$(child_1.subtree, child_2.subtree)$
14 **if** $rule \neq NULL$ **then**
15 **return** $rule$
16 **return** $\langle T_{src}, T_{tar} \rangle$

3.3.2 Rule Generation. This component aims at automatically deriving a translation rule from the simplified and translated queries.

Translation rule definition. We define the translation rule as mapping from a source pattern (Pat_{src}) to a target pattern (Pat_{tar}), where a pattern specifies an AST structure representing the syntactic form of the dialect. Let AST_{Q_s} and $AST_{Q_s'}$ be the ASTs of the simplified query and translated query, respectively. When AST_{Q_s} contains a tree structure matching the source pattern, it indicates a dialect-specific construct requiring transformation. SQL dialect translation is achieved by substituting segments of AST_{Q_s} that match the source pattern with the corresponding target pattern, and this process generates $AST_{Q_s'}$.

Figure 3 demonstrates the rule for mandatory alias assignment to derived tables in MySQL, which requires all derived tables (e.g., subqueries) to have explicit aliases. To facilitate representation, we utilize *bracket notation* [20] to express tree structures. As illustrated in Figure 3 (a), this notation employs nested parentheses to represent parent-child relationships between nodes, thereby uniquely identifying a tree. To extract the rule, we identify the minimal differing subtrees T_{src} in AST_{Q_s} and T_{tar} in $AST_{Q_s'}$, in which T_{src} and T_{tar} share the same root nodes, and the remaining parts of the ASTs (i.e., AST_{Q_s}.remove(T_{src}) and $AST_{Q_s'}$.remove(T_{tar})) are identical. As shown in Figure 3 (b), the minimal differing subtrees (i.e., T_{src} and T_{tar}) correspond to the trees rooted at the `table_ref` node. We perform the translation of dialects by leveraging the transformations on these two subtrees. Specifically, the subtree T_{src}, extracted from AST_{Q_s}, serves as the source pattern of the rule, while the other subtree T_{tar}, extraced from $AST_{Q_s'}$, constitutes the target pattern.

Rule extraction. As outlined in Algorithm 1, `RISE` automatically extracts the initial rule through the following steps:

- We utilize the function `getMinDiffTree()` to extract the minimal differing subtrees from AST_{Q_s} and $AST_{Q_s'}$, and use them as the source and target patterns for the initial rule (Line 1).
- In `getMinDiffTree()`, We first check whether the trees derived by removing T_{src} and T_{tar} from the AST_{Q_s} and the $AST_{Q_s'}$,

respectively, are identical (Line 3). If they differ, the function directly returns NULL, indicating that the current subtrees T_{src} and T_{tar} do not meet the requirements (Line 4).

- If the remaining parts of the trees are identical, it indicates that the current subtrees T_{src} and T_{tar} meet the requirements. To identify any smaller differing subtrees within them, we verify whether all child nodes of the root nodes in T_{src} and T_{tar} are identical. If any pair of child nodes mismatches (Line 6–10), T_{src} and T_{tar} are the minimal differing subtrees, and we return them directly (Line 11).
- Otherwise, we recursively call the function on their corresponding child subtrees (Line 12–13). If a non-empty result is returned, it is the minimal differing subtree pair, and we return it immediately (Line 14–15). If all recursive calls return empty values, T_{src} and T_{tar} are identified as the minimal differing subtrees and are directly returned (Line 16). In Figure 3, this process successively identifies the following pairs of subtrees with roots at: <root, root>, <simple_select, simple_select>, <from_clause, from_clause>, and <table_ref, table_ref>.

However, the extracted initial rule in Figure 3 exhibits limited applicability and struggles to be extended to a broader range of queries containing the same dialect. The inclusion of dialect-irrelevant contents in the rule adversely affects its matching conditions. Specifically, "(SELECT) (1)" in the initial source pattern will restrict the rule to only match cases where the subquery is "SELECT 1". Any variation in the content of the subquery renders the rule inapplicable. However, the content of the subquery is irrelevant to the dialect-specific operation of assigning an alias to the subquery. Regardless of the subquery's internal structure, the alias assignment remains unaffected. This highlights the necessity of abstracting dialect-irrelevant content within the rule. Additionally, all SQL statements require identifiers, such as table names and column names, to be unique within their respective scopes. In this example, the rule consistently assigns the same alias 't1' to the subquery. If multiple subqueries in an SQL statement require aliases, this rule would inevitably cause naming conflicts. Therefore, it is also essential to abstract the names of newly generated identifiers, ensuring that each is assigned a unique, randomly generated value.

Rule abstraction. As outlined in Algorithm 2, RISE uses DFS to traverse all subtrees (Line 17–18), checking if the conditions for three types of abstraction are met.

- *Node-level abstraction.* We assume that the names of identifiers and the values of constants are dialect-irrelevant. To normalize the patterns, we replace identical identifier names or constant values in both patterns with the symbol $N (where N is a subscript), which represents an arbitrary value of the same type (Line 5–7, 10–12). For instance, in the dialect translation "FETCH FIRST 100 ROWS ONLY → LIMIT 100", the integer value 100 is irrelevant to the dialect. Whether the value is 100 or 10, it does not affect the translation of this dialect. Therefore, we replace it with $1 to represent an arbitrary value of the integer, and gain "FETCH FIRST $1 ROWS ONLY → LIMIT $1".
- *New identifier abstraction.* If the target pattern contains newly introduced identifiers, we replace them with ANYVALUE, which denotes a randomly generated unique value (Line 8–9). For instance, in Figure 3, the identifier t1 is replaced with ANYVALUE.

Algorithm 2: Initial Translation Rule Abstraction

Input: Initial translation rule $\langle Pat^0_{src}, Pat^0_{tar} \rangle$
Output: Final translation rule $\langle Pat_{src}, Pat_{tar} \rangle$

```
1  Pat_src ← Pat^0_src
2  Pat_tar ← Pat^0_tar
3  abstractRule(Pat^0_tar.root)
4  Function abstractRule(curNode)
5      if isIdentifier(curNode) then
6          if Pat_src.contains(curNode) then
7              replaceAll(Pat_src, Pat_tar, curNode, $N)
8          else
9              replaceAll(Pat_src, Pat_tar, curNode, ANYVALUE)
10     else if isConst(curNode) then
11         if Pat_src.contains(curNode) then
12             replaceAll(Pat_src, Pat_tar, curNode, $N)
13     else if isNonTerminal(curNode) then
14         if Pat_src.contains(curNode.subtree) then
15             replaceAll(Pat_src, Pat_tar, curNode.subtree, TREE_N)
16         else
17             foreach childNode ∈ curNode.getChildren() do
18                 abstractRule(childNode)
```

Algorithm 3: Rule-Driven SQL Conversion

Input: Source query's AST AST_{Q_c}, translation rule $\langle Pat_{src}, Pat_{tar} \rangle$
Output: Converted query's AST $AST_{Q_c'}$

```
1  AST_Qc' ← AST_Qc
2  foreach subTree ∈ AST_Qc do
3      if patternMatch(subTree, Pat_src) then
4          newSubTree ← instantiate(subTree, Pat_src, Pat_tar)
5          replaceSubTree(AST_Qc', subTree, newSubTree)
6  Function patternMatch(subTree, Pat_src)
7      /* Abstract nodes: $N and TREE_N */
8      if isAbstractNode(Pat_src.root) ∧ Pat_src.root.getType() =
         subTree.root.getType() then
9          return True
10     else if Pat_src.root ≠ subTree.root then
11         return False
12     else
13         num ←
             max(Pat_src.root.childNum, subTree.root.childNum)
14         for i ← 0; i < num; i ← i + 1 do
15             child_1 ← subTree.root.getChild(i)
16             child_2 ← Pat_src.root.getChild(i)
17             if ¬ patternMatch(child_1.subtree, child_2.subtree) then
18                 return False
19         return True
```

- *Tree-level abstraction.* We assume that subtrees that remain unchanged during dialect translation are dialect-irrelevant, so we replace them with TREE_N, which represents an arbitrary tree rooted at a node of a specific type (Line 13–15). For example, in Figure 3, the subquery "SELECT 1" is abstracted as TREE_1, which represents any subtree rooted at 'select_with_parents'.

This two-phase process extracts translation rules by identifying the minimal differing subtrees under the same node between the source and target ASTs. It further enhances the generality of these rules by abstracting dialect-irrelevant content within the patterns.

3.4 Rule-Driven SQL Conversion

In this step, we utilize the generated rules to perform SQL dialect translation. As outlined in Algorithm 3, we search the AST of the source query for subtrees that match the rule's source pattern. If a matching subtree is found, it is replaced according to the target pattern. This process involves the following steps:

- *Source pattern matching.* We traverse all subtrees in the original source query AST AST_{Q_c} to identify those matching the source pattern (Line 2–3). During the matching phase (Line 6–19), we use DFS to traverse the input subtree and the source pattern, checking whether they are identical (Line 10–19). If the root node of source pattern is an abstract node (i.e., `$N` and `TREE_N`), we consider the match successful as long as the type of the root node in subtree matches the type of the root node in source pattern, skipping further recursion (Line 8–9).
- *Instantiation.* We replace abstract symbols in the target pattern (e.g., `$N`, `TREE_N`) with values matched from the source pattern and substitute `ANYVALUE` with a random unique value. This step yields a concrete subtree (Line 4).
- *Replacement.* We replace the subtree matched by the source pattern with the instantiated new subtree, thereby completing the dialect translation (Line 5).

By following these steps, we ensure accurate identification and conversion of dialect-specific structures within SQL queries, facilitating seamless translation across different SQL dialects.

4 EVALUATION

To evaluate the effectiveness of `RISE`, we raise the following four research questions:

- **RQ3 (Accuracy comparison):** How does `RISE` compare against state-of-the-art baselines in terms of accuracy?
- **RQ4 (Query reduction effectiveness):** How effective is query reduction in terms of accuracy and valid rule generation?
- **RQ5 (Efficiency comparison):** How does `RISE`'s efficiency compare to other LLM-based methods?
- **RQ6 (Generated rule analysis):** What translation rules can `RISE` generate?

4.1 Experimental Setup

4.1.1 Benchmarks. Since existing experimental data for evaluating LLMs on SQL dialect translation often fails to capture the complexity of real-world queries [42], we adopt TPC-DS [18] and SQLProcBench [24] (described in Section 2.1.1) as our benchmarks.

4.1.2 Baselines & Metrics. We compare `RISE` against the following six baselines. Since Mallet [30] is not publicly available, we are unable to include it as one of our baselines. We utilize accuracy to evaluate the translation capability of `RISE`, which is consistent with the metric described in Section 2.1.3.

- **SQLGlot:** SQLGlot [16] is an open-source SQL parser and transpiler that applies manually crafted rules to translate queries across dialects.
- **SQLines:** SQLines [17] automates schema and SQL migration between DBMSs via rule-based query translation.
- **jOOQ:** jOOQ [11] is a Java library for type-safe SQL construction, with built-in dialect translation from Java code.
- **`LLMTranslator`:** `LLMTranslator` employs a CoT strategy for dialect translation using two distinct LLM configurations: one with DeepSeek-V3 ($\texttt{LLMTranslator}_{DS}$) and another with GPT-4o ($\texttt{LLMTranslator}_{GPT}$). We present the details in Section 2.1.2.
- **CrackSQL:** CrackSQL [41] uses LLMs with cross-dialect embeddings and a local-to-global strategy to translate queries by syntactic segmentation and matching.

4.1.3 Implementation. In the following, we provide implementation details and LLM settings to facilitate reproducibility.

- **Adaptive parser building.** Since PostgreSQL [15] serves as the source DBMS for all experiments, we construct our adaptive parser based on PostgreSQLLexer.g4 and PostgreSQLParser.g4 grammar files [3], and build it with ANTLR version 4.13.2 [2].
- **Construction of TPC-DS.** The official TPC-DS benchmark [18] lacks PostgreSQL templates. DB2 queries [7] are the most similar, with 79/99 compiling on PostgreSQL (vs. 12 on MySQL). After manual fixes, we create a PostgreSQL-compatible TPC-DS version in which all queries execute successfully on PostgreSQL, while only 12 compile and run successfully on MySQL. This adjusted dataset forms our TPC-DS benchmark.
- **LLM settings.** All LLMs used in `RISE` are **GPT-4o** [10], a highly advanced LLM. For SQL reduction, the LLM is configured with a temperature of 0.7 and a token limit of 2048 to ensure comprehensive outputs. In `LLMTranslator`, the LLM used for feature summary is configured with a temperature of 0 and a token limit of 2048 to ensure deterministic outputs, while the LLM used for translation is configured with a temperature of 0.7 and the same token limit. Both `RISE` and `LLMTranslator` are set to a maximum of three iterations. To ensure the reliability of the results, we conduct repeated experiments on `RISE` to mitigate the randomness introduced by LLMs.

4.2 Accuracy Comparison

We conduct a comprehensive comparison of `RISE` against six baselines (i.e., SQLines [17], SQLGlot [16], jOOQ [11], $\texttt{LLMTranslator}_{DS}$, $\texttt{LLMTranslator}_{GPT}$ and CrackSQL [41]) by using two real-world benchmarks TPC-DS [18] and SQLProcBench [24]. On TPC-DS, we translate queries from PostgreSQL [15] to MySQL [12], while on SQLProcBench, we translate stored procedures from PostgreSQL to Oracle [14]. For the three SQL dialect translation tools, we rely on their online interfaces to manually translate queries. For CrackSQL, we utilize its open-source implementation, and adopt the default configuration and knowledge base to ensure a fair comparison.

4.2.1 Experimental Evaluation. Figure 2 shows the accuracy results of `RISE` and selected baselines in SQL dialect translation. Our experiments reveal the following three-fold key findings.

`RISE` demonstrates superior performance compared to all the baselines on the two benchmarks. Overall, `RISE` achieves an accuracy of 97.98% on TPC-DS and 100.00% on SQLProcBench. Notably, it outperforms the best baseline $\texttt{LLMTranslator}_{GPT}$ on TPC-DS by 7.78%, and outperforms the best baseline $\texttt{LLMTranslator}_{DS}$ on SQLProcBench by 22.22%. These improvements clearly underscore the effectiveness of `RISE` in SQL dialect translation task.

LLM-based methods achieve better results than traditional automated tools overall. Directly using LLMs for SQL dialect

Table 2: Translation Accuracy across Different Methods

Dataset	jOOQ	SQLines	SQLGlot	LLMTranslator$_{DS}$	LLMTranslator$_{GPT}$	CrackSQL	RISE
TPC-DS	73.74%	67.68%	75.76%	82.83%	90.91%	80.81%	**97.98%**
SQLProcBench	13.64%	0.00%	0.00%	81.82%	77.27%	4.55%	**100.00%**

```
1: CREATE table t1 (c1 CHAR(50));
2: CREATE table t2 (c1 INTEGER);
3: SELECT  t1.c1 FROM t2, t1
4: GROUP BY ROLLUP(t1.c1); -- result {null}
```

```
1: CREATE table t1 (c1 CHAR(50));
2: CREATE table t2 (c1 INTEGER);
3: SELECT  t1.c1 FROM t2, t1
4: GROUP BY (t1.c1) WITH ROLLUP; -- result {}
```

Figure 4: Differences in ROLLUP Semantics between MySQL and PostgreSQL.

translation (i.e., LLMTranslator$_{DS}$ and LLMTranslator$_{GPT}$) yields average accuracies of 86.87% on TPC-DS and 79.55% on SQLProcBench, demonstrating the promise of LLM-based methods. Besides, CrackSQL achieves an accuracy of 80.81% on TPC-DS but performs poorly on SQLProcBench, with an accuracy of only 4.55%. This poor performance is attributed to its lack of knowledge about stored procedures. In contrast, the three automated SQL dialect translation tools achieve around 70% accuracy on TPC-DS and struggle with the more complex SQLProcBench. Traditional tools depend on manually-crafted rules, which are often incomplete and error-prone. These limitations highlight the need for more advanced and adaptable translation mechanisms to ensure effective and accurate SQL dialect translation.

Advanced SQL features (e.g., stored procedures) significantly influences the accuracy of traditional SQL dialect translation tools. Notably, on the SQLProcBench benchmark, existing tools achieve near-zero accuracy due to the absence of necessary translation rules for processing stored procedures.

4.2.2 Bad Case Breakdown. We further conduct an in-depth investigation into the common error categories observed in the incorrect SQL dialect translations generated by RISE and the selected baselines. Specifically, we manually analyze the root causes of each incorrect translation case and categorize them accordingly. Our analysis reveals the following findings based on statistical results.

Error analysis for RISE. Only two queries from TPC-DS fail due to semantic inconsistencies. For instance, the symbol '||' acts as a logical OR operator in MySQL but is used for string concatenation in PostgreSQL. However, due to the empty result set, RISE does not automatically detect this semantic inconsistency and fails to translate them. Aside from these exceptions, RISE successfully translates all other queries.

Error analysis for traditional SQL dialect translation tools. Our analysis categorizes the failures of traditional tools on the TPC-DS benchmark into three types: **parsing errors**, **incorrect translation rules**, and **missing translation rules**. For instance, jOOQ experiences parsing errors with complex queries (e.g., query 23 exceeding 100 lines), highlighting its limitations with complicated SQL features. Regarding incorrect translation rules, SQLGlot introduces errors when handling 'ORDER BY' clauses, while jOOQ encounters issues translating the 'INTERVAL' function. In terms of the missing translation rules, SQLGlot lacks support for assigning aliases to derived tables, jOOQ is unable to process 'FULL OUTER JOIN', and SQLines is deficient in handling the 'INTERVAL' function, 'NUMERIC' type conversion, and 'FULL OUTER JOIN' operation. Notably, SQLines occasionally fails to translate the 'FETCH FIRST n ROWS' clause, revealing a lack of robustness in its rule for this feature. As demonstrated in Figure 4, the 'ROLLUP' operation exhibits slight semantic differences between MySQL and PostgreSQL, and none of the three traditional tools process 'ROLLUP' correctly, indicating a broader challenge in handling advanced SQL features. It is noteworthy that RISE effectively addresses these issues with its LLM-assisted translation rule generation module. By leveraging the extensive domain knowledge of LLM, RISE automatically generates translation rules that overcome the limitations of traditional tools in rule adequacy.

Error analysis for LLM-based methods. As analyzed in Section 2.3, the errors in LLMTranslator primarily arise from LLM hallucinations, including incorrect modifications of dialect-irrelevant content and improper handling of dialect-specific constructs. RISE addresses these challenges via dialect-aware query reduction. CrackSQL also exhibits significant limitations, largely due to its heavy reliance on a knowledge base and poor handling of stored procedures. Its performance on SQLProcBench is severely impacted by missing stored procedure information, with 68.18% of errors caused by incorrect "Cannot translate!" judgments. On TPC-DS, CrackSQL encounters 19 issues: 1 judgment error, 2 crashes ("list index out of range" and "max_tokens is too large"), 2 syntax errors, and 14 semantic inconsistencies—all stemming from LLM hallucinations. For example, CrackSQL incorrectly assumed MySQL does not support INTERSECT (contradicting the official documentation [13]) and replaced it with UNION. By removing dialect-irrelevant elements (e.g., INTERSECT), RISE prevents LLMs from introducing such errors, while query reduction further improves accuracy on dialect-specific constructs.

> ***Answer to RQ3:*** *RISE significantly outperforms the six baselines in terms of accuracy, underscoring its effectiveness in translating SQL dialects.*

4.3 Query Reduction Effectiveness

To assess the impact of Query Reduction (QR) on RISE, we conduct an ablation study on TPC-DS and SQLProcBench. The study evaluates how removing the QR module affects RISE's translation accuracy (Acc) and the number of valid rules (NVR) generated.

4.3.1 Results Analysis. As shown in Table 3, removing the QR module (i.e., w/o QR) significantly impacts the overall performance of RISE. Specifically, without reduction, direct processing of source

Table 3: Ablation Study Results

Model	TPC-DS		SQLProcBench	
	Acc	NVR	Acc	NVR
RISE	97.98%	13	100.00%	37
w/o QR	81.82% (16.49% ↓)	3 (76.92% ↓)	72.73% (27.27% ↓)	20 (45.95% ↓)

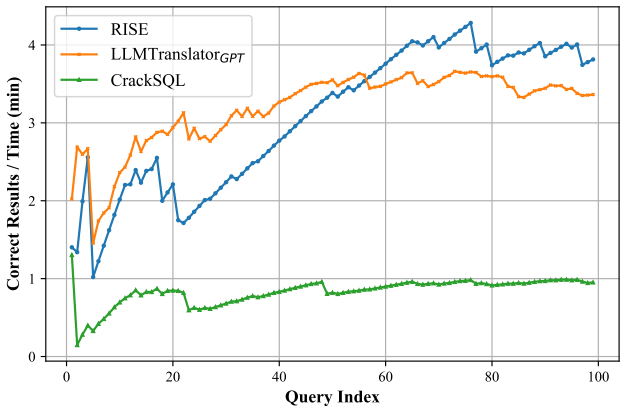


Figure 5: Efficiency Comparison of Correct Translations per Unit Time Between RISE and LLM-based methods.

queries reduces accuracy by 16.49% (TPC-DS) and 27.27% (SQLProcBench), and the number of generated valid rules drops by 76.92% and 45.95%, respectively. The necessity of QR is twofold.

Protection against undesirable modifications. Without the QR module, the LLM may inadvertently modify dialect-irrelevant elements during the translation process. Even if such modifications do not alter the final query's equivalence, they can limit the general applicability of the generated rules. To illustrate, consider a scenario where an LLM unnecessarily adds an 'AS' between table names and aliases—a common tendency in its outputs. If a rule r is to translate dialect d, this extra 'AS' will limit r to queries in d that lack 'AS' before aliases. More critically, erroneous modifications (e.g., the example of transforming 'SUM(SUM())' into 'SUM()' discussed in Section 2.3) can compromise query equivalence without triggering compiler warnings, making automated detection of such semantic inconsistencies extremely challenging.

Reduction in query length. The QR module simplifies the source SQL queries by removing dialect-irrelevant elements. As observed in **O3**, LLMs struggle to process lengthy and complex queries. By simplifying the queries, the QR module enhances the accuracy of LLMs in translating SQL dialects.

***Answer to RQ4**: SQL reduction significantly improve the performance of SQL dialect translation in terms of accuracy and valid rule generation.*

4.4 Efficiency Comparison

RISE has been shown to achieve higher accuracy than other methods. In light of this, we further investigate the translation efficiency of RISE in comparison with other LLM-based methods (i.e., LLMTranslator$_{GPT}$ and CrackSQL [41]) on TPC-DS [18]. To ensure fairness, all methods use the same LLM, GPT-4o [10]. The benchmarking metric is defined as follows:

Correct translations per unit time. Let n denotes the number of correctly translated queries, and t denotes the total translation time in minutes. The efficiency metric is then measured by computing the ratio $\frac{n}{t}$. This metric represents the number of correct translations per minute, providing a quantitative measure to compare the efficiency of SQL dialect translation.

As shown in Figure 5, since RISE starts with an empty rule set and has higher complexity, it takes longer to generate valid rules, resulting in lower efficiency compared to LLMTranslator$_{GPT}$. However, after accumulating sufficient rules—around query 57—it applies them directly for translation, achieving a stable rate of 3.78 correct translations per minute, surpassing LLMTranslator$_{GPT}$ at 3.3 translations per minute, which processes each query independently. The performance gap is expected to grow with larger datasets. However, due to the high complexity of CrackSQL, its efficiency has consistently been the lowest. Finally, the total time for RISE, LLMTranslator$_{GPT}$, and CrackSQL to complete 99 translations is 25.42 minutes, 26.75 minutes, and 84.02 minutes, respectively.

***Answer to RQ5**: Compared to other LLM-based methods,* RISE *ultimately achieves higher efficiency.*

4.5 Generated Rule Analysis

As shown in Table 3, RISE produces a total of 13 rules for TPC-DS [18] and 37 rules for SQLProcBench [24]. All of the generated rules are confirmed to be valid through manual inspection.

On TPC-DS, RISE generates 6 rules to handle dialect-specific constructs: FULL OUTER JOIN, INTERVAL, CAST AS NUMERIC/INT, derived table aliases, and FETCH FIRST N ROWS. On SQLProcBench, it produces 22 rules covering type conversions (INT, CHAR, DECIMAL, VARCHAR, temporal types) and stored procedure constructs (e.g., declarations, cursors, returns, exceptions, IF-ELSE, assignments, LIMIT). Below are two simplified rules generated during evaluation.

Listing 2: INTERVAL Function Translation Rule

```
1 source pattern:(...(A_expr_add(TREE_1)(+)(...(
      Constinterval(INTERVAL))(Sconst(Anysconst('60␣
      days'))))))
2 target pattern:(...(Type_function_name(Identifier(
      DATE_ADD))...(A_expr_add(TREE_1)))(,)(
      Func_arg_expr(A_expr(INTERVAL)(Interval_value(
      Iconst(60))(Time_unit(DAY))))))
```

Listing 2 presents a rule generated by RISE in TPC-DS for translating the INTERVAL function, which existing tools like jOOQ [11] and SQLines [17] fail to handle correctly. While PostgreSQL requires time values in INTERVAL to be quoted (e.g., '60 days'), MySQL forbids quotes around such values. However, a simple string-matching approach risks modifying unrelated string literals resembling time values. RISE overcomes this using the rule-driven SQL conversion

algorithm (Algorithm 3) to accurately identify INTERVAL parameters and apply correct conversions.

Listing 3: Procedure Declaration Conversion Rule

```
source pattern:(Createfunc_opt_item(LANGUAGE)(plpgsql
    )(AS)(Func_as(Plsql_block($$)...($$))))
target pattern:(Createfunc_opt_item(IS)(Func_as(
    Plsql_block ...)))
```

Listing 3 shows a rule translating PostgreSQL stored procedures to Oracle by removing "LANGUAGE plpgsql", replacing "AS $$" with "IS", and omitting the closing "$$" delimiter. Specifically, PostgreSQL uses "LANGUAGE plpgsql" to specify the procedural language and employs "$$" as the procedure body delimiter, whereas Oracle omits the language specification and uses "IS" to begin the procedure body without closing delimiters. For example, a PostgreSQL procedure declared as "CREATE OR REPLACE PROCEDURE ... LANGUAGE plpgsql AS $$ ... $$" becomes "CREATE OR REPLACE PROCEDURE ... IS ..." in Oracle, ensuring syntactic compatibility while preserving the procedure's logic.

***Answer to RQ6**: On both TPC-DS and SQLProcBench,* RISE *generates numerous and diverse effective rules.*

5 DISCUSSION

In this section, we discuss the threats to validity and the limitations of our approach.

5.1 Threats to Validity

The primary threat to the validity of RISE lies in the diversity of the SQL queries used for evaluation and its generalizability to other DBMSs. In this paper, we collect a total of 143 SQL queries from TPC-DS [18] and SQLProcBench [24], ensuring a reasonable degree of diversity and quality in our evaluation. For comparison, we select three traditional SQL dialect translation tools [11, 16, 17] as well as three LLM-based approaches: CrackSQL [41] and our experimental prototype, $\texttt{LLMTranslator}_{DS}$ as well as $\texttt{LLMTranslator}_{GPT}$. In future work, we aim to expand the benchmarks for a more comprehensive evaluation of RISE's generalizability across diverse DBMSs and emerging SQL dialects.

5.2 Limitations

Existence of semantic inconsistency. Currently, RISE cannot fully resolve semantic inconsistencies, as relying solely on the execution results of DBMSs cannot ensure complete accuracy. For example, as discussed in 4.2.2, RISE misidentified the '||' operator, which denotes string concatenation in PostgreSQL and logical OR in MySQL. This error introduced a semantic inconsistency between the source and translated queries, leading to translation failure. Moreover, since the data failed to meet the query predicate conditions, both DBMSs returned empty results, causing RISE to mistakenly conclude that the queries were semantically equivalent. It is challenging to automatically validate the semantic equivalence of the source queries and their corresponding translated queries across DBMSs. Resolving semantic inconsistency in the SQL dialect translation task remains as our future work.

Inability to handle dialects with ambiguous semantics. Certain SQL dialects exhibit semantic ambiguity, as their behavior depends on schema information from the database and cannot be determined solely from the SQL query itself. For example, the division operator ('/') in PostgreSQL has two possible semantics: integer division and floating-point division. The specific behavior depends on the operands: if the operands include floating-point numbers, it represents floating-point division; otherwise, it represents integer division. Since the translation rules currently defined do not account for schema information, RISE is unable to robustly translate queries such as SELECT t1.c1 / t2.c2, resulting in potentially unreliable translation rules. In the future, we plan to extend the rules to incorporate schema information.

6 RELATED WORK

SQL dialect translation based on manually-crafted rules. Most current SQL dialect translation tools [5, 11, 16, 17] are rule-driven, leveraging the limited and deterministic nature of SQL dialect grammars. A straightforward strategy to address recurring error cases is to design new rules, which can then be integrated into transformation tools to support comprehensive SQL migrations. However, this method has a clear limitation: the rules must be manually constructed. This process is not only time-consuming but also demands a deep understanding of the SQL dialects involved.

LLM-based SQL dialect translation. LLMs have shown remarkable code-writing capabilities [31], providing a promising solution for SQL dialect translation by reducing manual effort. Mallet [30] leverages LLMs to automatically generate translation rules, thereby reducing the dependence on manual rule construction for unsupported SQL dialects. However, the generated rules tend to be relatively simplistic, limiting Mallet's ability to handle complex SQL features such as stored procedures. Additionally, Mallet cannot perform SQL dialect translation independently; it relies on the traditional tool SQLGlot [16] for execution, restricting its applicability to only those DBMSs already supported by SQLGlot. To mitigate such issues, a recent approach called CrackSQL [41] introduces a more structured framework. It segments complex queries into smaller, functionally coherent units and applies function normalization and irrelevant query abstraction to simplify translation, thereby improving robustness and reducing errors. However, the reliability of LLM-generated outputs remains a significant concern [31, 35]. For example, although CrackSQL [41] incorporates LLMs with a knowledge base for validation, our results reveal frequent syntax errors and semantic inconsistencies. These findings highlight the limitations of relying solely on LLMs for accurate SQL translation.

Program reduction. Many approaches have been proposed to address program reduction from different perspectives. HDD [29] and T-PDD [34] leverage hierarchical structures and probabilistic modeling to guide the reduction process. LPR [39] introduces LLMs to improve flexibility and automation. Perses [33] and T-Rec [36] utilize formal and lexical grammar guidance to ensure syntactic correctness during reduction. In contrast, PPR [40] explores a distinct approach through pairwise program reduction. Among these, PPR is less suitable for SQL query reduction due to its reliance on pairwise comparisons, which conflict with the single-query nature

of SQL. The other approaches are more applicable and offer potential to improve SQL reduction rates. While existing database testing methods [21, 32, 37] primarily rely on simplistic AST random deletion, incorporating these techniques into SQL reduction represents a significant research direction.

7 CONCLUSION

In this paper, we begin by exploring the capabilities and limitations of LLMs in SQL dialect translation through an empirical study. Based on our observations, we propose a novel approach, RISE, for efficiently and accurately translating SQL dialects across different DBMSs. Specifically, we first apply a dialect-aware query reduction technique to remove dialect-irrelevant elements from the source query. We then develop an LLM-assisted translation rule generation algorithm that automatically creates translation rules based on the simplified query, enabling effective translation of SQL dialects regardless of query complexity. Comprehensive experiments on two real-world benchmarks demonstrate the effectiveness of RISE.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62302493), Basic Research Project of ISCAS (ISCAS-JCZD-202403), Major Project of ISCAS (ISCAS-ZD-202302), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044).

References

[1] Amazon DMS. (Tool). https://aws.amazon.com/dms/schema-conversion-tool/. Accessed: 2025-01.
[2] ANTLR. (Tool). https://www.antlr.org/. Accessed: 2025-01.
[3] ANTLR Grammars. (Grammar). https://github.com/antlr/grammars-v4. Accessed: 2025-03.
[4] Azure DMS. (Tool). https://learn.microsoft.com/zh-cn/azure/dms/dms-overview. Accessed: 2025-03.
[5] Datometry. (Tool). https://datometry.com/. Accessed: 2025-01.
[6] DB-Engines Ranking. (Website). https://db-engines.com/en/ranking. Accessed: 2025-03.
[7] DB2. (DBMS). https://www.ibm.com/products/db2-database. Accessed: 2025-01.
[8] DeepSeek-V3. (Model). https://github.com/deepseek-ai/DeepSeek-V3. Accessed: 2025-05.
[9] Dow Jones. (Website). https://aws.amazon.com/cn/solutions/case-studies/dow-jones/. Accessed: 2025-01.
[10] GPT-4o. (Model). https://openai.com/index/hello-gpt-4o/. Accessed: 2025-01.
[11] jOOQ. (Tool). https://www.jooq.org/translate/. Accessed: 2025-01.
[12] MySQL. (DBMS). https://dev.mysql.com/doc/refman/8.0/en/. Accessed: 2025-01.
[13] MySQL. (Documentation). https://dev.mysql.com/blog-archive/intersect-and-except-in-mysql-80/. Accessed: 2025-05.
[14] Oracle. (DBMS). https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/. Last accessed on 2025-03.
[15] PostgreSQL. (DBMS). https://www.postgresql.org/docs/15/. Accessed: 2025-01.
[16] SQLGlot. (Tool). https://github.com/tobymao/sqlglot. Accessed: 2025-01.
[17] SQLines. (Tool). https://www.sqlines.com/. Accessed: 2025-01.
[18] TPC-DS Benchmark. (TPC). https://www.tpc.org/tpcds/. Accessed: 2025-01.
[19] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1097–1107.
[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. The MIT Press.
[21] Ziyu Cui, Wensheng Dou, Yu Gao, Rui Yang, Yingying Zheng, Jiansen Song, Yuan Feng, and Jun Wei. 2025. Simple Testing Can Expose Most Critical Transaction Bugs: Understanding and Detecting Write-Specific Serializability Violations in Database Systems. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 2547–2560.
[22] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting Hallucinations in Large Language Models using Semantic Entropy. *Nature* 630, 8017 (2024), 625–630.
[23] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 243–252.
[24] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding their Usage in the Wild. *Proceedings of the VLDB Endowment (PVLDB)* 14, 8 (2021), 1378–1391.
[25] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proceedings of the ACM on Programming Languages (PACMPL)* 8, OOPSLA1 (2024), 132:1–132:29.
[26] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 1243–1246.
[27] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured Chain-of-Thought Prompting for Code Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 2 (2025), 37:1–37:23.
[28] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 743–754.
[29] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 142–151.
[30] Amadou Latyr Ngom and Tim Kraska. 2024. Mallet: SQL Dialect Translation with LLM Rule Generation. In *Proceedings of International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (AI4DM)*. 5.
[31] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 2 (2025), 1–28.
[32] Jiansen Song, Wensheng Dou, Yingying Zheng, Yu Gao, Ziyu Cui, Wei Wang, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 2281–2294.
[33] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 361–371.
[34] Guancheng Wang, Yiqian Wu, Qihao Zhu, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2023. A Probabilistic Delta Debugging Approach for Abstract Syntax Trees. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE)*. 763–773.
[35] Justin D. Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection Not Required? Human-AI Partnerships in Code Translation. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*. 402–412.
[36] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. 2025. T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 2 (2025), 31.
[37] Rui Yang, Ziyu Cui, Wensheng Dou, Yu Gao, Jiansen Song, Xudong Xie, and Jun Wei. 2025. Detecting Schema-Related Logic Bugs in Relational DBMSs via Equivalent Database Construction. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1725–1747.
[38] Wenhua Yu, Zhenhao Chen, Haibo Liu, and Yongbin Bai. 2024. Research on Localized Translation of SQL Dialect. In *Proceedings of Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. 1322–1326.
[39] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 261–273.
[40] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 338–349.
[41] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Translation System. *Proceedings of International Conference on Management of Data (SIGMOD)* 3, 3 (2025), 26.
[42] Ran Zmigrod, Salwa Alamir, and Xiaomo Liu. 2024. Translating between SQL Dialects for Cloud Migration. In *Proceedings of IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*. 189–191.