Cracking SQL Barriers: An LLM-based Dialect Translation System

WEI ZHOU, Shanghai Jiao Tong University, China
YUYANG GAO, Tsinghua University, China
XUANHE ZHOU, Shanghai Jiao Tong University, China
GUOLIANG LI, Tsinghua University, China

Automatic dialect translation reduces the complexity of database migration, which is crucial for applications interacting with multiple database systems. However, rule-based translation tools (e.g., SQLGlot, jOOQ, SQLines) are labor-intensive to develop and often (1) fail to translate certain operations, (2) produce incorrect translations due to rule deficiencies, and (3) generate translations compatible with some database versions but not the others.

In this paper, we investigate the problem of automating dialect translation with large language models (LLMs). There are three main challenges. First, queries often involve lengthy content (e.g., excessive column values) and multiple syntax elements that require translation, increasing the risk of LLM hallucination. Second, database dialects have diverse syntax trees and specifications, making it difficult for cross-dialect syntax matching. Third, dialect translation often involves complex many-to-one relationships between source and target operations, making it impractical to translate each operation in isolation. To address these challenges, we propose an automatic dialect translation system `CrackSQL`. First, we propose *Functionality-based Query Processing* that segments the query by functionality syntax trees and simplifies the query via (i) customized function normalization and (ii) translation-irrelevant query abstraction. Second, we design a *Cross-Dialect Syntax Embedding Model* to generate embeddings by the syntax trees and specifications (of certain version), enabling accurate query syntax matching. Third, we propose a *Local-to-Global Dialect Translation* strategy, which restricts LLM-based translation and validation on operations that cause local failures, iteratively extending these operations until translation succeeds. Experiments show `CrackSQL` significantly outperforms existing methods (e.g., by up to 77.42%). The code is available at *https://github.com/weAIDB/CrackSQL*.

CCS Concepts: • **Information systems → Structured Query Language**.

Additional Key Words and Phrases: Database Dialect Translation, Large Language Models

ACM Reference Format:
Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Translation System. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 141 (June 2025), 26 pages. https://doi.org/10.1145/3725278

1 Introduction

Dialect translation aims to transform a SQL query from one database system (e.g., PostgreSQL) into a functionally equivalent query executable on a target database system (e.g., MySQL). This capability is crucial for alleviating database migration efforts and supporting applications such as

Authors' Contact Information: Wei Zhou, Shanghai Jiao Tong University, China, weizhoudb@gmail.com; Yuyang Gao, Tsinghua University, China, 21373016@buaa.edu.cn; Xuanhe Zhou, Shanghai Jiao Tong University, China, zhouxh@cs.sjtu.edu.cn; Guoliang Li, Tsinghua University, China, liguoliang@tsinghua.edu.cn.

ACM 2836-6573/2025/6-ART141
https://doi.org/10.1145/3725278

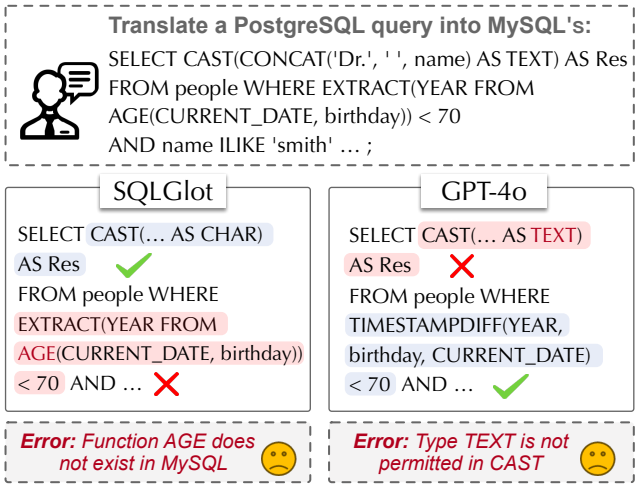

Fig. 1. Translation Example. Existing methods fail to accurately conduct the two translations in the query[1].

NL2SQL [22], ORM frameworks [13], ETL processes [13], and cross-database analytics tools [20], enabling seamless interaction with diverse database systems.

Rule-based Dialect Translation. Despite the importance of dialect translation, automating this process is challenging, where the dialects of different databases involve unique syntax rules, reserved keywords, built-in functions, etc. Consequently, existing tools (e.g., SQLGlot [7], jOOQ [3], SQLines [8]) rely on human engineers to manually identify pairs of equivalent syntax variations and formalize them into translation rules. However, these tools often struggle to accurately translate queries in many cases.

(1) Missing Translations: Some query operations should be translated but are not. For instance, as shown in Figure 1, when translating the example query [26] from PostgreSQL to MySQL, the AGE function in PostgreSQL, which calculates the difference between two dates is unsupported in MySQL but cannot be translated by existing translation tools like SQLGlot [7].

(2) Incorrect Translations: Some query operations are translated incorrectly. For instance, SQLGlot [7] converts the function JSON_EXTRACT_PATH from PostgreSQL into JSON_EXTRACT for Oracle, which does not exist. Notably, there are over 200 issues in SQLGlot's code repository that have reported incorrect translations caused by problems like improper rule implementation [7].

(3) Version Incompatibility: Translation tools have poor database version support, i.e., the translated query operations are valid for specific database versions but not others. For instance, when translating from PostgreSQL to Oracle, SQLGlot (without the version selection option) translates the query operation "LIMIT 1" into "FETCH FIRST 1 ROW ONLY". This translated operation is valid for Oracle 19c, but unsupported for Oracle 11g.

Traditionally, addressing these problems requires human engineers to write extensive code to handle rule deficiencies and special cases (e.g., ***2k+*** issues remain open in the code repository of jOOQ [3]), which is both costly and inflexible, especially when accommodating updates such as the upgrades of dialect versions.

LLM-based Dialect Translation. Recently, large language models (LLMs) have demonstrated strong semantic understanding and coding capabilities [25, 33, 44]. They can generate SQL queries (e.g., NL2SQL [22, 24]), perform query rewrite [29], and interpret documents for tasks like knob

[1]See detailed real-world queries successfully translated by `CrackSQL` at *https://github.com/weAIDB/CrackSQL*.

tuning [23, 45, 46] and database diagnosis [48]. This suggests that LLMs could be a promising solution to the limitations of rule-based database tools.

However, compared with common translation tasks [24, 37], dialect translation poses unique challenges for LLMs to address and ***directly leveraging LLMs remains impractical***. First, LLMs often struggle to capture the subtle syntax differences between database dialects and fail to accurately memorize the syntax specifications, which are critical for accurate translation. For instance, as shown in Figure 1, GPT-4o fails to translate the query because, without specifying the syntax constraints, GPT-4o cannot recognize that the TEXT data type is not allowed in CAST function of MySQL. Second, LLMs may inadvertently replace functions with those of different functionalities or output formats. For instance, Llama3.1 translates the MySQL query "DATE_FORMAT(CURRENT_TIMESTAMP(), '%Y')" into the Oracle query "TO_CHAR(SYSDATE, 'RR')", where '%Y' returns a four-digit year ("2024") while 'RR' provides a two-digit result ("24"). Additionally, dialect translation lacks large-scale open dataset of typical SQL translation pairs, unlike other programming language translations such as *Java* ⇔ *C#*[2].

To address the limitations of both rule-based translation tools and LLM, we propose a new dialect translation paradigm. First, to mitigate the error-prone characteristic, we *limit LLM's involvement* in the translation process. That is, rather than indiscriminately using LLM to translate entire SQL statements, we focus on specific incompatible operations. Second, to address adaption challenges like supporting new dialects, we incorporate dialect-specific knowledge (e.g., valid syntax and specifications in the target database dialect) to enhance LLM translation.

Challenges. To realize the above dialect translation paradigm, there are three main challenges.

C1: How to segment complex and lengthy queries? Queries often involve lengthy content (e.g., excessive column values in IN clause) and multiple syntax elements that require translation (e.g., the translations of CAST and AGE functions in the example query of Figure 1), increasing the risk of dialect translation failures.

C2: How to match functionally equivalent syntax across diverse database dialects? Dialect translation needs to identify syntax elements with equivalent functionalities across different database dialects in consideration of both the syntax tree structures and specifications, typically written in various styles and lengths, making syntax functionality matching a tricky task.

C3: How to conduct many-to-one dialect translation? Two key sub-challenges remain in dialect translation. First, it is crucial to select as few operations as possible for LLM translation to minimize the risk of hallucinations. Second, many-to-one translations are inherently complex (e.g., mapping multiple operations into one in the target dialect), and determining which operations should be translated as a whole is challenging. For instance, as shown in Figure 1, the EXTRACT and AGE operations in PostgreSQL need to be processed together by the LLM, so as to be correctly translated into TIMESTAMPDIFF in MySQL.

Our Methodology. To address these challenges, we propose an automatic dialect translation system (`CrackSQL`). First, we introduce *Functionality-based Query Processing*, which segments the input query according to the syntax trees of different functionalities and simplifies the segmented operations with customized function normalization and irrelevant query abstraction (for **C1**). Second, to identify syntactic elements from the target dialect, we develop a *Cross-Dialect Syntax Embedding Model*, which effectively calculates functionality embeddings based on both syntax tree structures and specifications (for **C2**). Additionally, we employ a *Retrieval-Enhanced Contrastive Learning* to effectively train the model using limited positive samples and noisy negative samples. Third, to achieve accurate dialect translation, we propose a *Local-To-Global Translation Strategy*, which focuses on operations that cause local failures (e.g., incompatibilities or those that cannot be

[2]https://github.com/microsoft/CodeXGLUE

translated using local single syntax element) through query validation, utilizes translation tools and target-syntax-augmented LLM to translate these operations, and iteratively extends these operations for re-translation if translation failures occur (for **C3**).

Contributions. In summary, we make the following contributions:
(1) We design an automatic dialect translation system that effectively translates SQL queries for different dialects (see Section 3).
(2) We propose *Functionality-based Query Processing* that separates and simplifies complex queries to reduce the translation difficulty and facilitate accurate translations (see Section 4).
(3) We propose a *Cross-Dialect Syntax Embedding Model* that aligns the embeddings of equivalent syntax variants across dialects, enabling more accurate target dialect syntax matching (see Section 5).
(4) We propose a *Local-to-Global Dialect Translation* strategy that utilizes target-syntax-augmented LLM to extensively translate problematic operations that cause local failures (see Section 6).
(5) Our extensive experiments on real-world datasets demonstrate significant translation accuracy improvements over existing methods (see Section 7).

2 Preliminaries

2.1 Database Dialect and Incompatibility

Database Dialect refers to the specific implementation of SQL standards in a database. Syntax differences between database dialects can cause *incompatibility issues*, where a SQL query written for one database may fail to execute correctly on another. Incompatibilities typically arise from differences in the following five aspects.

(1) Syntax Rules: Databases use distinct syntax rules for common operations such as column and table annotations and definitions about join and data manipulation. For example, a subquery without an alias is allowed in Oracle, while MySQL requires that each subquery must be assigned an alias.

(2) Keywords: Databases contain a set of reserved keywords to support different operations. For example, PostgreSQL has the keyword FULL OUTER JOIN for fetching all the rows from the joined tables, while MySQL can only obtain the same functionality combined with the two keywords LEFT OUTER JOIN and RIGHT OUTER JOIN. Besides, the keyword NULLS LAST in PostgreSQL specifies the order of null values, which is also not supported in MySQL.

(3) Functions and Operators: Databases implement a series of (customized) functions and operators. For example, MySQL develops a function "LAST_DAY()" to return the last day of a month, while PostgreSQL can utilize the "DATE_TRUNC()" function and "+" and "-" arithmetic operators to achieve the same functionality.

(4) Data Types: Databases may different names for the same or similar data types. For example, MySQL leverages DATETIME to annotate the date and time type, while Oracle adopts TIMESTAMP.

(5) Others: Databases may handle constraints (e.g., foreign keys, unique columns) and system settings differently. For instance, MySQL on Linux is case-sensitive by default (with lower_case_table_names=0), while PostgreSQL usually converts table and column names to lowercase, making it case-insensitive.

Note that we use *syntax elements* to represent keywords, functions and operators in the following sections without ambiguity.

2.2 Database Dialect Translation

Database Dialect Translation refers to translating a query that causes incompatibility issues by following the standards of target database dialect to ensure compatibility and *equivalent functionality* while executing in the target database system.

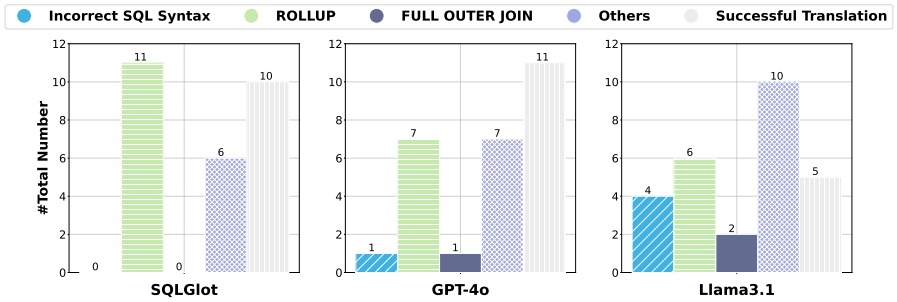

Fig. 2. Error Distribution of Translation from PostgreSQL to MySQL for TPC-DS queries (*Others* denote queries that yield different results due to database inner differences, e.g., precision of numeric values).

Definition 2.1 (Functional Equivalence)**.** A query operation q_i^T, which is an implementation of syntax S_i^T, in database D^T is *functionally equivalent* to a query operation q_i^O in database D^O if it produces the same execution results or has the same effect as q_i^O.

Example 1. In PostgreSQL, the operation CONCAT('hello', NULL) returns "hello", while in MySQL, CONCAT('hello', NULL) returns NULL. These operations are not functionally equivalent. To achieve the same result, the MySQL operation should include a conditional expression, such as CASE ... WHEN.

Example 2. Within CREATE clause, NUMERIC(10, 2) in PostgreSQL is equivalent to NUMBER(10, 2) in Oracle, both defining a numeric column with up to 10 digits, including 2 decimal places.

Definition 2.2 (Database Dialect Translation)**.** Given a query Q^O written in a source database dialect, Q^O is composed of one or more operations $\{q_i^O\}$. *Database Dialect Translation* refers to the process of mapping each operation q_i^O to one or more functionally equivalent operations in the target database dialect. The translated query Q^T must (1) strictly follow the target dialect syntax S^T and (2) maintain functional equivalence to Q^O.

Example 3. As shown in Figure 1, translating a PostgreSQL query that *retrieves the names of people younger than 70* into an equivalent MySQL-compatible query requires separately translating both the CAST and EXTRACT operations. Specifically, we first resolve the data type incompatibility by replacing TEXT with CHAR in the CAST operation. Then we modify the EXTRACT operation to account for MySQL's lack of support for the AGE function.

In this paper, we mainly consider the common translation types found in existing dialect translation tools [3, 7, 8], i.e., translating (1) syntax rules, (2) keywords, (3) built-in functions and operators, (4) data types across different dialects. Note that dialect translation and query rewrite [29, 43] are two different problems. (1) *Query Rewrite* transforms queries under the same dialect, while *Dialect Translation* converts SQLs of different syntax among different dialects. (2) *Query Rewrite* modifies query structures or remove redundant operations for better performance; while *Dialect Translation* adjusts queries to match another dialect's syntax to resolve incompatibilities.

In this paper, we research how to process queries into normalized operations with lower translation complexity (see Section 4), how to match equivalent syntax elements in the target database dialect (see Section 5), and how to accurately translate queries from the local operation level to the global query level (see Section 6).

2.3 Limitations of Existing Methods

To better understand the limitations of existing methods and motivate the design of our system, we conducted an experiment (migrating TPC-DS queries from PostgreSQL to MySQL) that reveals the problems of SQLGlot [7], GPT-4o, and Llama3.1 [42] in dialect translation. As shown in Figure 2, we have three observations.

(Observation 1) Translation Affects Only Small Portions of Lengthy Complex Queries. Our first observation is that the query operations requiring translation occupy only small parts of the lengthy TPC-DS template queries. Specifically, the number of words in the experimented template queries that require translation is 131 on average, while we notice that only the ROLLUP expressions in most of the template queries require translation (i.e., GROUP BY ROLLUP(col_list) → GROUP BY col_list WITH ROLLUP). This observation motivates us to *focus translation efforts on certain query operations that require functional equivalence across dialects.*

(Observation 2) Failure Due to Lack of Translation Rules and Knowledge. As shown in Figure 2, ROLLUP and FULL OUTER JOIN account for a large proportion of translation failures. Specifically, SQLGlot, GPT-4o, and Llama3.1 incur translation errors, i.e., 40.74%, 29.63% and 29.63% respectively, because they fail to handle the translation of ROLLUP and FULL OUTER JOIN. This indicates that *existing rule-based translators lack certain translation rules, and LLMs are unaware of functionally equivalent specifications*, both of which are crucial for successful translation.

(Observation 3) Error-Prone Characteristics of LLMs. We observe that LLMs are prone to making mistakes due to the hallucination problem. Specifically, both GPT-4o and Llama3.1 might rename columns to ones that do not exist in the database (e.g., mistakenly changing "`ca_state`" to "`ca.state`" based on their semantic understanding). Furthermore, Llama3.1 makes syntax mistakes by appending the SELECT clause into the WITH clause as the last common table expression called "final_result", which leads to the absence of a SELECT clause. It indicates *the necessity of enhancing the reliability of LLMs when applied in dialect translation with strict grammar requirements.*

3 System Overview

To address the above problems in dialect translation, we propose an automatic dialect translation system (`CrackSQL`), which is composed of *Functionality-based Query Processing*, *Cross-Dialect Syntax Embedding*, and *Local-To-Global Dialect Translation.*

Functionality-based Query Processing. LLMs have significant hallucination problems when handling queries with lengthy content and complex structures, especially for those requiring the translation of multiple syntax elements. Thus, we first propose a *tree-based strategy* to segment the original query Q^O by the syntax trees of different functionalities in the source dialect (i.e., the whole query is transformed into a set of simpler operations $\{q'_i\}$). Next, we further simplify the query via rules like *operation normalization* (e.g., converting customized functions into common ones) and *operation abstraction* (e.g., replacing translation-irrelevant parts like column and table names with non-terminal expressions). The entire query processing procedure is *automatic* by parsing and traversing the SQL and matching the prepared specifications with the BNF grammar parser in a depth-first order.

Cross-Dialect Syntax Embedding. Another challenge is how to match equivalent syntax elements across various target database dialects. To this end, we design a *Cross-Dialect Embedding Model* to calculate the functionality embeddings for syntax elements, including (1) code structure encoding (for various syntax elements in the syntax tree), (2) Mixture-of-Experts (MoE) based syntax specification encoding (for specifications with diverse styles and lengths), and (3) syntax-specification aggregation (for aligning specification embeddings with the corresponding syntax element embedding).

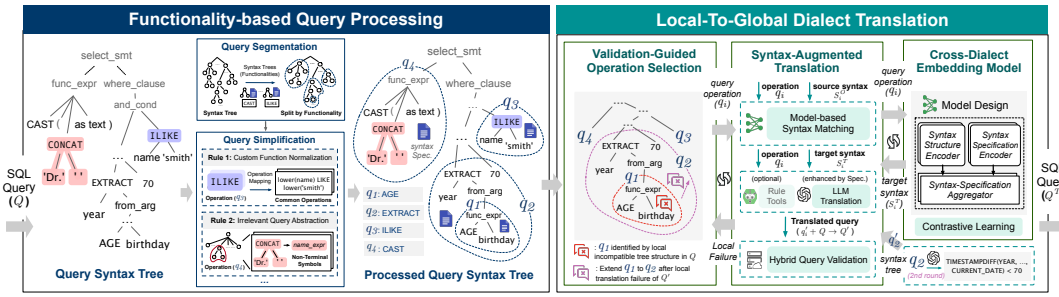

Fig. 3. System Overview of CrackSQL.

Local-To-Global Dialect Translation. To avoid unnecessary translations and enable simultaneous consideration of multiple syntax elements required in complex translation, this module translates the processed operations $\{q'_i\}$ to derive the ultimate query Q^T in the target database dialect by gradually extending query operations from the local level to the global level. At the *local operation level*, we identify operations $\{q'_j\}$ that cause local failures: ***(1) Incompatibilities*** identified by the BNF parser and ***(2) Insufficiencies*** recognized when translation fails using local syntax information (e.g., exceeding the maximum LLM trials). For operations with incompatibilities, we match the target dialect syntax for $\{q'_j\}$ using the *Cross-Dialect Embedding Model* and utilize a hybrid approach combining rule-based translation tools (e.g., SQLGlot) and LLM to translate these operations. For operations that are insufficient to translate, we iteratively extend the operation to include adjacent elements (q''_j). We continue translating using the above method until, after replacing the origin query with the translated operations, the query Q^T is syntactically valid and functionally equivalent to Q^O.

Note we only use LLM in translating some segmented SQL operations (corresponding to certain functionality in the prepared specifications), and resolve the LLM hallucination problem via hybrid validation mechanisms.

Offline Preparation. In the offline stage, we prepare syntax element specifications and train the embedding model: (1) We extract syntax elements from database parser files. Each syntax element is annotated with (i) syntax tree in a unified BNF format and (ii) functionality specification (e.g., from the official documents). Specifically, we analyze the organization and structure of the SQL syntax in the documents, which are typically well-maintained (e.g., PostgreSQL provides the grammar rules and usage for each released version[3]) and well-structured (e.g., in Oracle, each SQL syntax is a title, followed by usage descriptions in the underlying paragraphs[4]). Based on the analysis over document structure, we compose scripts[5] to automatically annotate syntax specifications. (2) To match syntax functionality accurately, we employ *Retrieval-Enhanced Contrastive Learning* to effectively train the *Cross-Dialect Embedding Model* with limited positive samples (i.e., equivalent syntax elements) and judiciously select negative samples, such as non-equivalent syntax elements with similar specifications (see Section 5). The generation of training samples is fully automated without human intervention (see Section 5). For instance, we utilize existing rule-based systems to produce positive examples that exhibit equivalent functionality for a given syntax element. Notably, the *Cross-Dialect Embedding Model* is trained only once, rather than separately for each translation task (e.g., MySQL to PostgreSQL), enabling it to support translations across various dialects available in CrackSQL.

[3] https://www.postgresql.org/docs/
[4] https://docs.oracle.com/cd/E11882_01/server.112/e41084/functions009.htm
[5] https://github.com/code4DB/CrackSQL

4 Functionality-based Processing

We properly process the input queries with various structures and operations, including syntax functionality annotation, tree-based query segmentation, and rule-based query simplification.

4.1 Syntax Functionality Annotation

Given a database dialect, we extract the syntax elements from its query parser files (e.g., the gram.y file in PostgreSQL). For each syntax element S, we require (i) the syntax tree in Backus-Naur Form (BNF) of S for exact syntax matching and (ii) the specification of S's functionality for subsequent translation (see Section 6).

(1) Syntax Tree Generation: Different database parsers follow diverse grammar rules, making their parsed syntax trees difficult to compare. To enable syntax matching across databases, we generate syntax trees in a unified BNF format. We maintain and use a series of BNF definitions [18] to standardize the syntax representation of queries. And, for every database, we derive the syntax trees of syntax elements in the database, where the leaf nodes represent BNF terminal symbols (e.g., SQL literals), and the internal nodes represent non-terminal symbols (e.g., SQL clauses or functions).

(2) Syntax Specification Annotation: For the BNF-based syntax tree of S, we automatically annotate functionality specifications by exact syntax element matching. We start by comparing the syntax tree nodes of S with those from syntax elements in sources like official documents (e.g., PostgreSQL, MySQL, and Oracle separately offer 245, 466, and 490 pages), using a depth-first traversal. Next, we annotate the matched subtrees (query operations) of S with their syntax trees and specifications, including (i) functionality, (ii) usage constraints, (iii) arguments and default values, and (iv) usage examples, all of which are vital for syntax matching and translation. For instance, since the keyword AS appears in various grammar rules (e.g., in CAST function or as table aliases), we differentiate specifications for AS by syntax trees.

4.2 Tree-based Query Segmentation

Next, we discuss how to segment queries based on the syntax trees of different functionalities. There are generally two strategies:

(1) Lexical-Based Segmentation uses regular expressions to split an SQL query based on specific keywords or patterns. Typical SQL keywords such as SELECT serve as markers to divide the query into smaller parts. However, it treats the query as a flat sequence of tokens, ignoring the underlying syntax and the hierarchical relationships in the query (e.g., subqueries within SELECT and WHERE clauses depend hierarchically on the outer query).

(2) Tree-Based Segmentation conducts an exact matching of sub-trees in the query's syntax tree for capturing the query structures.

Lexical-based segmentation struggles with complex queries (e.g., with nested subqueries). For these queries, the operations derived by lexical-based segmentation may incorrectly span across multiple clauses. Thus, we adopt tree-based segmentation. Similar to *syntax specification annotation*, we traverse syntax tree T^Q in a depth-first order to match the predefined syntax elements. We denote the exact matching function as $M(t)$, which returns TRUE if the substree t matches that of an element in the dialect syntax S, and FALSE otherwise. If a match is found, we extract the subtree t rooted at node $\mathcal{N}_k$. This subtree corresponds to the operation that begins at $\mathcal{N}_k$, such as the SELECT clause. The tokens contained in t are then converted back into SQL query, i.e., $q = \text{Unparse}(t)$. In this way, we ensure that all tokens within the subtree t belong to the same operation (e.g., all tokens in the SELECT clause are kept together).

After syntax matching, we aggregate these identified operations, i.e., $Q = \{q_j \mid M(t) = \text{True}\}$. Each operation q_j forms a valid SQL expression annotated with functionality specifications (obtained in Section 4.1), which can be used in subsequent translation.

4.3 Rule-based Query Simplification

Based on the observation in Section 2, when translating SQL queries between different dialects, it is essential to simplify operations by (1) replacing customized functions (e.g., LAST_DAY() in MySQL) with equivalent common ones and (2) focusing on syntax elements that are crucial for accurate translation. Therefore, we employ predefined rules to further process operations segmented from the query Q, retaining only the translation-relevant components.

The core principle of this simplification is to ensure that within the syntax subtree of an operation, no nodes contain customized functions, and any non-leaf nodes that do not affect translation are replaced with predefined terms or query templates. To this end, we provide simplification rules in the three main scenarios:

• *Case 1: Simplifying Customized Functions.* Database systems often implement a variety of customized functions to enhance usability (e.g., the LAST_DAY function in MySQL). However, these functions are often unsupported or exhibit inconsistent behavior in other database systems. To tackle this issue, we maintain mappings (sourced from SQLGlot and human expertise) of customized functions to equivalent expressions using common functions (if available) for query normalization. For example, with the rule "str ILIKE pattern::=LOWER(str) LIKE LOWER(pattern)", we normalize PostgreSQL's case-insensitive ILIKE function into the standard SQL expression by applying LIKE with lowercase transformations.

• *Case 2: Abstracting Irrelevant Expressions.* Some functions or expressions nested within an operation q_i are generally not used when translating q_i. For instance, consider the operation "CAST(CONCAT('DR.', '', name) AS TEXT)", which is matched by the CAST syntax. The inner CONCAT function does not affect the translation of the CAST operation and can be abstracted away. As a result, the expression is simplified to CAST(column_expr AS TEXT), where column_expr represents a non-terminal symbol.

• *Case 3: Abstracting Irrelevant Query Clauses.* Certain query clauses may not be relevant to the primary function or behavior of a specific syntax and can be masked. For instance, in the MySQL query "SELECT * FROM child WHERE age >= 10 LIMIT 2 OFFSET 10", which is matched by the LIMIT syntax, the SELECT clause and WHERE condition are not relevant to the translation of the LIMIT clause. Thus, these parts can be masked, retaining only the LIMIT portion, i.e., select_stmt LIMIT 2 OFFSET 10.

5 Cross-Dialect Embedding Model

After query processing, another key challenge is matching equivalent syntax in the target database dialect (e.g., functions with identical functionality but different names). However, most existing general-purpose embedding models [16] train on paragraphs from the same documents and do not explicitly consider pairs of cross-dialect specifications. Consequently, they cannot accurately capture the functional similarities of syntax elements across diverse syntax rules and textual descriptions (see Figure 8).

To address this problem, in this section, we propose *Cross-Dialect Embedding Model*, which generates syntax functionality embedding based on both the syntax tree structure and syntax specification features. Additionally, we employ a retrieval-enhanced contrastive learning approach to effectively synthesize translation samples and train the model with judiciously selected samples (e.g., operation with different functionality but high similarity).

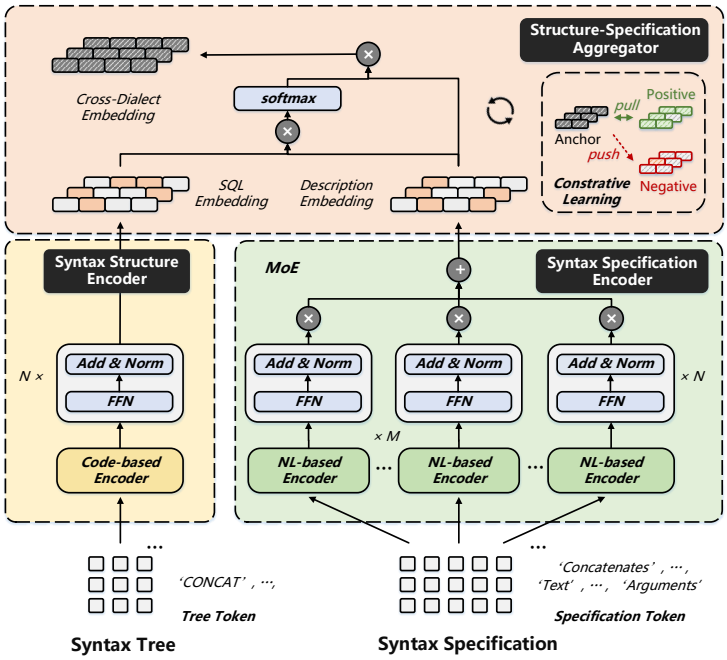

Fig. 4. Cross-Dialect Syntax Embedding Model.

5.1 Model Architecture

Given a query syntax S^Q, the goal of *Cross-Dialect Embedding Model* is to encode its functional characteristics by leveraging both syntactic information (i.e., syntax keywords and structures) and semantic information (i.e., textual specifications). This approach ensures that queries with the same functionality, despite having different syntax, keywords, or specifications, will produce similar embeddings and can be effectively matched.

Formally, *Cross-Dialect Embedding Model* is a function $E : \mathcal{S} \rightarrow \mathbb{R}^d$, where $\mathcal{S}$ is the space of all possible query syntax and d is the embedding dimension. The embedding $E(S^Q)$ captures the functionality of the syntax S^Q. In this paper, we design the model architecture in a hybrid way to capture different features, which is composed of three main modules:

Syntax Structure Encoder. This module aims to capture the structural features of the query syntax. We utilize a code-based encoder model, such as StarEncoder [28], which is pretrained on code data (including SQL queries) to extract a structural representation of the query. For the input query Q, we first tokenize it into a sequence of tokens $\mathbf{t} = [t_1, t_2, \ldots, t_n]$ (e.g., "CONCAT", "(", ... in Figure 4). We then feed the token sequence into the code-based encoder to obtain the structural embedding $\mathbf{h}_q^{\text{struct}} \in \mathbb{R}^{d_s}$:

$$\mathbf{h}_q^{\text{struct}} = \text{CodeEncoder}(\mathbf{t}) \tag{1}$$

The code-based encoder is trained to capture syntactic features of different syntax elements (e.g., the CONCAT() function in Figure 4), making it suitable for representing syntax structures.

Syntax Specification Encoder. Since different dialect specifications show diverse writing styles (e.g., the CONCAT() function in MySQL[6] is detailed with illustrative examples, while the one

[6]https://dev.mysql.com/doc/refman/8.4/en/string-functions.html#function_concat

in PostgreSQL[7] is succinct as one row), this module employs multiple natural languages (NL)-based encoders to capture multi-dimensional features of specifications and acquire a weighted representation of the specification in a mixture-of-experts (MoE) style.

Let $\mathbf{d}$ denote the textual specification associated with the query syntax S. We employ a set of K NL-based encoders $\{\text{NLEncoder}_k\}_{k=1}^{K}$, each capturing different aspects of the specification. For instance, some encoders can extract information from long passages, while others are skilled at representing short paragraphs [34]. Each encoders generates an embedding vector, i.e., $\mathbf{h}_{q,k}^{\text{desc}} = \text{NLEncoder}_k(\mathbf{d})(k = 1, \ldots, K)$.

We then aggregate these embeddings using a gating mechanism to obtain the final specification embedding $\mathbf{h}_q^{\text{desc}}$:

$$\mathbf{h}_q^{\text{desc}} = \sum_{k=1}^{K} \alpha_k \mathbf{h}_{q,k}^{\text{desc}} \tag{2}$$

where α_k are the gating weights determined by a gate function:

$$\alpha_k = \frac{\exp(g_k(\mathbf{d}))}{\sum_{j=1}^{K} \exp(g_j(\mathbf{d}))} \tag{3}$$

where $g_k(\mathbf{d})$ is the output of the gating network for the k-th encoders, allowing the model to dynamically assign weights based on the input specification. For instance, a higher weight for encoder specialized in understanding and encoding the detailed illustrations about the CONCAT function in MySQL.

Structure-Specification Aggregator. Given the syntax structure embedding $\mathbf{h}_Q^{\text{struct}}$ and the specification embedding $\mathbf{h}_Q^{\text{spec}}$, this module combines the two features using a multi-head cross-attention. It ensures that the model focuses on the most relevant parts of the specification in relation to the syntax structure, i.e.,

$$\mathbf{E}(q) = \text{CrossAttention}(\mathbf{h}_q^{\text{struct}}, \mathbf{h}_q^{\text{desc}}) \tag{4}$$

where $\text{CrossAttention}\ \mathbf{E}(q) = Concat(head_1, \ldots, head_h)\mathbf{W}^O$, each head $head_i = \text{Attention}(\mathbf{h}_q^{struct}\mathbf{W}_i^Q, \mathbf{h}_q^{desc}\mathbf{W}_i^K, \mathbf{h}_q^{desc}\mathbf{W}_i^V)$. The attention function is defined as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{5}$$

where $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, $\mathbf{W}_i^V$, and $\mathbf{W}^O$ are projection matrices, and d_k is the dimension of the keys.

In this way, we can align and integrate information from both embeddings, effectively filtering out irrelevant parts of the specification and focusing on syntax-relevant content. Continuing from previous examples, suppose the specification includes query syntax descriptions and usage hints. The cross-attention mechanism will assign higher attention weights to relevant parts (e.g., the example SQL query and its illustrations about CONCAT), integrating them with the structural embedding to produce the final embedding $\mathbf{E}(q)$.

In summary, the advantages are three-fold. *(1) Robustness to Syntax Variations:* The model incorporates the functional characteristics of different database dialects, making it resilient to variations in query syntax elements and rules. *(2) Adaptability to Diverse Specifications:* Using an MoE-style Syntax Specification Encoder, we can process specifications of varying lengths and styles by assigning appropriate weights to different NL-based encoders. *(3) Effective Feature Integration:* The multi-head cross-attention integrates only the most relevant descriptive features with syntax elements, improving the ultimate functionality embedding quality.

[7] https://www.postgresql.org/docs/14/functions-string.html

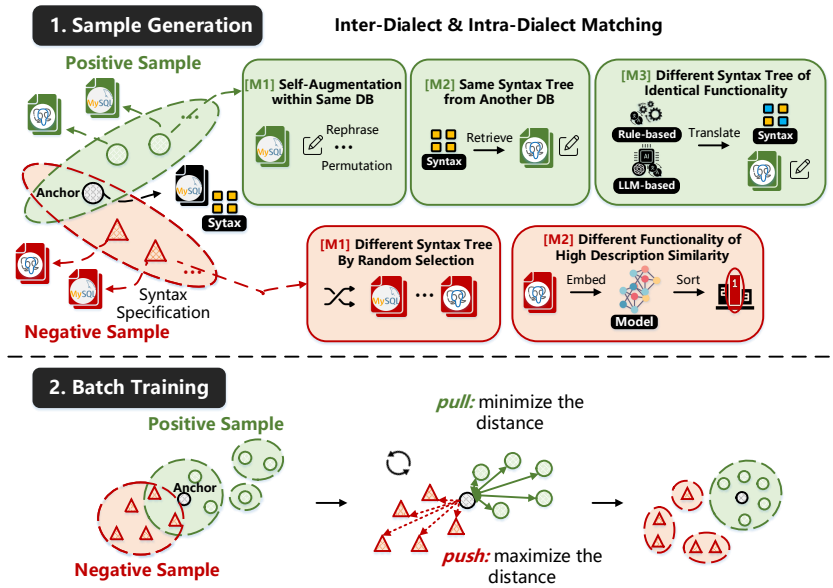

Fig. 5. Illustration of Retrieval-Enhanced Contrastive Learning (*Green:* Positive Sample; *Red:* Negative Sample).

5.2 Retrieval-Enhanced Contrastive Learning

Next, we explain how to effectively train *Cross-Dialect Embedding Model* with limited samples. Training this model requires a substantial number of positive and negative samples to distinguish between functionally equivalent syntax elements and those that are not [12]. Positive samples are pairs of syntax elements with the same functionality, while negative samples are pairs that differ functionally. However, despite the availability of numerous queries from different dialects, obtaining equivalent pairs based on their annotated specifications is challenging (e.g., there are 735 candidate PostgreSQL specifications and 532 MySQL specifications). To address this issue, as illustrated in Figure 5, we first introduce methods to automatically generate these samples as initial training data. We then present a novel retrieval-enhanced contrastive learning paradigm to train the model using the generated samples.

5.2.1 Training Sample Generation

For each database system, we collect query syntax trees and specifications from official syntax documentation (see Section 4.1). We represent each type of query syntax tree as a pair of syntax elements and specifications, i.e., $s_i = \langle Q_i^s, \mathbf{d}_i^s \rangle$, where Q_i^s denotes the query keywords or structure, and $\mathbf{d}_i^s$ represents the textual specification. For each syntax element (an anchor element), we generate positive and negative samples based on the following strategies:

Positive Sample Generation . As shown in Figure 5, we synthesize positive samples using the following methods.

M1: *Self-Augmented Specifications.* For a syntax s_i, we perform self-augmentation on its specification $\mathbf{d}_i^s$ by rephrasing sentences, replacing words with synonyms, and permuting sentence order [32]. Each augmented specification $\mathbf{d}_i^{s\prime}$ forms a new syntax pair $\langle Q_i^s, \mathbf{d}_i^{s\prime} \rangle$. The original syntax pair and each augmented pair form positive sample pairs, reinforcing the model's robust understanding of varying expressions of the same functionality [14, 46].

M2: *Cross-Database Specifications.* For a syntax s_i, we search for syntax s_j in other databases that has the same keywords of identical functionality but different specifications. For example, built-in functions in both MySQL and PostgreSQL may have different specification styles discussed in Section 5.1. Pairs (s_i, s_j) form positive samples, helping the model recognize equivalent functionalities across different database dialects.

M3: *Tool-Based Translations.* For a syntax s_i, we utilize existing dialect translation tools (e.g., SQLGlot) to translate the query Q_i^s of syntax s_i into other database dialects, producing new queries Q_k^s. Each translated query combined with its corresponding specification forms a new syntax $s_k = \langle Q_k^s, \mathbf{d}_k^s \rangle$. The pairs (s_i, s_k) serve as positive samples.

Negative Sample Generation . After generating positive samples, we cluster syntax elements based on functional equivalence, i.e., syntax elements that are positive samples with each other are grouped into the same cluster. To construct negative samples, we consider pairs of syntax elements from different clusters whose size is large than one, which are expected to be functionally different.

By employing above methods (e.g., using deterministic tools and basic synonym transformations), we can generate high-quality training samples with minimal noise. Moreover, if any noisy samples lead to inaccurate translations (i.e., mismatching target syntax by the embedding model), such errors can still be detected through the validation mechanism in *Local-To-Global Translation*.

5.2.2 Contrastive Model Training

Given the generated samples, contrastive model training aims to (1) reduce the distances between embeddings of positive samples (pulling them closer), and (2) increase the distances between embeddings of negative samples (pushing them apart). For any syntax s_i as an anchor sample, we obtain its positive samples $\mathcal{P}(i)$ and negative samples $\mathcal{N}(i)$. The *Cross-Dialect Embedding Model* encodes each syntax s into an embedding vector $\mathbf{h}_s = f_\theta(s)$ and is trained once and works for translations across different dialects supported.

Hard-Negative Sample Selection. Within the set $\mathcal{N}(i)$, many negative samples are either (1) already well-distinguished from s_i's embedding (i.e., with low cosine similarity), or (2) introduce significant noise due to their dissimilarity or irrelevance. To alleviate the efforts spent on such ineffective samples, we introduce **hard negative samples**, i.e., samples that (1) have different functionality from s_i but (2) possess embeddings similar to that of s_i. For example, the specification of function A is redundantly referred to by the specification of function B, which leads to the matching for function A mistakenly occurring for function B. To derive the set of hard negative samples $\mathcal{N}'(i)$, we first select negative samples whose embeddings are more similar to s_i than those of any positive samples (i.e., they are ranked higher in similarity).

Loss Function Design. Using the samples in $\mathcal{P}(i)$ and $\mathcal{N}'(i)$, we design the contrastive loss function for s_i as:

$$\mathcal{L}_i = -\frac{1}{|\mathcal{P}(i)|} \sum_{s_p \in \mathcal{P}(i)} \log \left(\frac{\exp(z_{ip})}{\exp(z_{ip}) + \sum_{s_n \in \mathcal{N}'(i)} \exp(z_{in})} \right)$$

where z_{ij} denotes the scaled cosine similarity between embeddings $\mathbf{h}_{s_i}$ and $\mathbf{h}_{s_j}$, i.e., $z_{ij} = \text{sim}(\mathbf{h}_{s_i}, \mathbf{h}_{s_j})/\tau$. τ is a temperature hyper-parameter controlling the concentration level of the distribution. z_{ip} is the scaled similarity of s_i with a positive sample, and z_{in} is the scaled similarity of s_i with a hard negative sample.

This loss function encourages the model to maximize the similarity between the anchor sample and its positive samples while minimizing the similarity with hard negative samples.

Algorithm 1: LocalToGlobalDialectTranslation

Input: Origin Query Q; *source_dialect*; *target_dialect*

Initialize the translated query Q' as Q;
repeat
  incompatible_ops = **DetectIncompatibility**(Q', *target_dialect*);
  foreach *op in incompatible_ops* **do**
    repeat
      translated_op = **Translate**(*op*, *target_dialect*);
      if *translated_op is not None* **then**
        Replace *op* with *translated_op* in Q';
      else
        Extend *op* to include more operations in Q';
    until *translated_op is not None* ***or*** *cannot extend op*;
until *no incompatibility warnings* ***or*** *arriving maximal trials*;
if Q' *is valid* ***and*** *equivalent to* Q **then**
  return Q' ; // Translation successful
return "Cannot Translate" ; // Translation failed

Function DetectIncompatibility(Q, *target_dialect*)

Input: Query Q; *target_dialect*

Generate the syntax tree of Q in *target_dialect* BNF definitions;
Collect incompatibility warnings;
Identify the operations causing each warning;
return list of operations causing incompatibility warnings;

Function Translate(*op*, *target_dialect*)

Input: Operation to translate *op*; *target_dialect*

Match *op* with relevant syntax in *target_dialect*;
Use LLM to translate *op* within maximal trials;
if *translated_op is valid* ***and*** *equivalent to op* **then**
  return *translated_op*
return None ; // Translation failed

6 Local-To-Global Translation

To translate queries with various syntax elements, in this section we propose Local-To-Global translation (see Algorithm 1) that achieves two main objectives: (*i*) avoids unnecessary translations of SQL operations that are already compatible with the target dialect, and (*ii*) supports complex translations, such as those involving multiple origin SQL operations to achieve a functionality within the target dialect (e.g., translating the combined EXTRACT and AGE function in PostgreSQL into the TIMESTAMPDIFF function in MySQL).

Validation-Guided Operation Selection. Since not all the query operations require translation (discussed in Section 2), we identify query operations that are (1) incompatible with the target database dialect or (2) cannot be correctly translated with its local syntax tree and specification.

(1) Incompatibility Warning Detection. We try to generate the syntax tree for the input query Q based on the BNF definitions in the target dialect. Any query operation that raises incompatibility

warnings (identified by the BNF grammar parser) during the tree generation is flagged to be translated. For instance, consider the PostgreSQL SQL in Figure 3, the "CAST(CONCAT('Dr.', '') AS TEXT)" casts the result of CONCAT function into the TEXT type. However, the TEXT type is not allowed in the CAST function of MySQL. Thus, the query operation of CAST function raises an error and is flagged.

(2) Operation Selection. The operations raising the incompatibility warnings are isolated as translation units. Continuing the above example, the problematic query operation is the CAST function. We select this operation for exclusive translation. Operations such as different built-in functions are typical candidates for isolation.

(3) Operation Extension. For the selected operation q_i, if the LLM fails to translate q_i within maximal trials (i.e., having parsing errors in the revised query Q'), we denote the current operation q_i as insufficient and dynamically extend the operation to include adjacent operations that might assist the translation. This extension allows for a wide scope of translation. For example, if the CAST function is not successfully translated within maximal trials, we expand it to the select_stmt to perform translation over a wider scope with more query operations involved.

Syntax-Augmented Translation. For each identified problematic operation, we employ LLM informed by specifications via model-based syntax matching to perform translation. This ensures the translated operations are syntactically correct in the target dialect and maintain the original query's intent (functionality equivalent).

(1) Model-Based Syntax Matching: For each identified operation Q_i, we employ the *Cross-Dialect Embedding Model* to align syntax elements in the target dialect (see Section 5.2). For instance, the CAST function in PostgreSQL is mapped to the CAST function in MySQL. Then, both the specifications about CAST function in PostgreSQL and MySQL serve as the input to instruct LLM for accurate translations.

(2) Iterative LLM Translation: Given the input SQL, its specification, and the matched specifications in the target dialect, we employ LLM (e.g., GPT-4o) to perform iterative translations. In each iteration, the LLM considers both the operation and its matched syntax elements to progressively refine the translation.

(3) Hybrid Query Validation: We adopt a hybrid strategy to validate the translated operations. We design a two-step validation mechanism to utilize local failures and specifications to perform syntactically correct and functionally equivalent translations.

• ***Step 1: Syntactic Validation.*** The translated operations replace the identified ones to derive the translated query Q'. Then, we generate the syntax tree of Q' with the BNF definitions in the target dialect and observe whether incompatibility warnings persist.

• ***Step 2: Semantic Validation.*** Once the translated query Q' is syntactically correct (i.e., no incompatibility error exists), we instruct LLM to reflect whether the translation is successful. Our intuition to adopt LLMs for semantic validation is that they showcase good code understanding to write SQL queries [22]. Specifically, we provide LLM with the specifications of the identified and the translated query operations as well as the input query Q and translated query Q', instructing its reasoning about the equivalence judgment of two queries (typically easier than conduct translation).

The validation is conducted between the above syntactic and semantic validation in multiple rounds until no error is detected. With this two-step validation, we can effectively evaluate the functionality equivalence of the translated query (see Table 1) and do not need to actually execute in the database (which is generally not allowed in real scenarios).

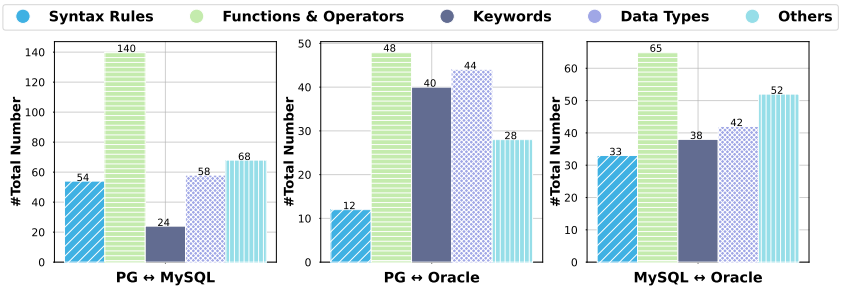

Fig. 6. Translation Type Distribution in Our Benchmark (A single query may involve multiple types).

7 Experiments

In this section, we first introduce a dialect translation dataset collected from mainstream sources. Then, we use the dataset to conduct extensive experiments to evaluate the performance of `CrackSQL` and compare it with state-of-the-art methods.

7.1 Experimental Setup

Baselines. We compare three popular translation tools and LLMs in the experiments: ***(1) SQLGlot*** [7] is an open-sourced translation engine that supports common database dialects and translation rules (e.g., it defines 13, 16, 22 data type translation rules for MySQL, PostgreSQL, and Oracle, respectively); ***(2) jOOQ*** [3] is an open-sourced translation engine that integrates empirical rules like SQLGlot; ***(3) SQLines*** [8] is a close-sourced translation tool; ***(4) LLMs:*** We also evaluate several LLMs, including GPT-4o [2], `CodeLlama-7B` [1], and `Llama3.1-8B-Instruct` [4]. `CrackSQL` utilizes GPT-4o as the translation model by default, and the other two LLMs are evaluated in Section 7.5.3. Each LLM performs dialect translations based on the given problem instructions[8].

Evaluation Metric. We adopt three evaluation metrics: ***(1) Executable Ratio*** (Acc_{EX})***:*** The ratio of the translated queries that are executable in the target database without raising incompatibility error (e.g., incorrect data types or functions); ***(2) Result Accuracy*** (Acc_{RES})***:*** The ratio of the translated queries that return the strictly identical results in the target database as the origin queries in the source database, including the returned data format, precision, and displayed order; ***(3) Retrieval Precision:*** The success ratio of syntax specification retrieval by embedding models (see Section 7.5.2).

Implementation. The tested database systems include MySQL v8.0, PostgreSQL v14, Oracle 11g. We utilize the vector database ChromaDB to conduct syntax specification retrieval. The workstation setup is two Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz, 256 GB main memory, and four GeForce RTX 3080 Ti graphics cards.

Hyper Parameters. For *Cross-Dialect Embedding Model*, we adopt StarEncoder pretrained over the SQL corpus as *syntax structure encoder*; and bge-large-en-v1.5, all-MiniLM-L6-v2, stella_en_400M_v5 from the MTEB leaderboard as *syntax specification encoders*, with only few parameters required to be fine-tuned. The number of retrieved syntax specifications k equals 3 by default. The fine-tuned LLMs adopt LoRA by default; and hyper-parameters like learning rate and training epoch are set to 1e-4 and 50 respectively.

7.2 Dialect Translation Dataset

Existing database performance benchmarks [10, 11, 19] cover limited syntax elements, with few query templates suitable for dialect translation. For instance, in TPC-H [11], only queries with

[8]The detailed LLM instruction prompts are provided at *https://github.com/weAIDB/CrackSQL*.

Table 1. Translation Accuracy (%) of Different Methods ('-' denotes translation is not supported in Ora2Pg).

Method	PG → MySQL		MySQL → PG		PG → Oracle		Oracle → PG		MySQL → Oracle		Oracle → MySQL	
	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
SQLGlot [7]	74.19	70.97	60.32	60.32	55.81	53.49	53.85	46.15	29.27	20.73	73.33	66.67
jOOQ [3]	70.97	70.97	39.68	39.68	62.79	60.47	84.62	53.85	47.56	35.37	80.0	53.33
Ora2Pg [5]	-	-	33.33	33.33	-	-	76.92	46.15	-	-	-	-
SQLines [8]	9.68	9.68	31.75	31.75	53.49	48.84	61.54	38.46	39.02	32.93	80.0	60.0
GPT-4o [2]	61.29	61.29	50.79	44.44	60.47	55.81	84.62	53.85	12.2	10.98	80.0	73.33
CrackSQL (Ours)	**87.1**	**74.19**	**85.71**	**79.37**	**69.77**	**67.44**	**92.31**	**61.54**	**59.76**	**42.68**	**93.33**	**80.0**

timestamp expressions (e.g., the INTERVAL keyword) are applicable for dialect translation. And the dozens of translatable queries in TPC-DS [10] are insufficient for comprehensive testing. Therefore, we spent around 6 human months preparing a dataset of typical dialect translation pairs from the following sources:

• **Open-Source NL2SQL Benchmark.** We collect translation pairs from the open-source NL2SQL benchmarks, which consist of complex queries reflecting heterogeneous user questions daily. For instance, we collect the queries provided in the BIRD benchmark [27], where each user question is associated with queries written for both MySQL and PostgreSQL. We de-duplicate the queries originating from the same query template (i.e., only differ in the parameters) and expand these queries to Oracle during our experiments (i.e., verify and collect the successfully translated queries).

• **Public Code Repository.** We collect translation pairs from actively maintained Github repositories (e.g., SQLGlot, jOOQ) in two ways: (1) We extract and collect the queries from the relevant GitHub issues (e.g., the ones including the translation keywords), which are well-formatted in the SQLGlot repository. (2) We extract the test cases involved in the code of repositories like SQLGlot.

• **Online Code Website.** We turn to the dumped data of websites like StackExchange [9] to extract the queries from the questions tagged with PostgreSQL / MySQL / Oracle. Since questions on these websites are typically lengthy with multiple responses, we utilize LLM (i.e., GPT-4o) to help us extract the relevant queries.

Dataset Statistics. *(1) Translation Samples:* From above sources, we gain ***248***, ***142*** and ***111*** query pairs for PG ↔ MySQL, PG ↔ Oracle and MySQL ↔ Oracle respectively[9]. As shown in Figure 6, it showcases distinct features for the dialect translation problem, including the typical translation types discussed (see Section 2). *(2) Syntax Specifications:* Moreover, we collect ***524***, ***735***, and ***332*** functionality specifications from parser files and documents for MySQL, PostgreSQL, and Oracle, respectively (see Section 4.1).

7.3 Performance Comparison

With the translation dataset, we separately test CrackSQL and the baselines in six translation scenarios of the three database systems.

Translation Accuracy. As shown in Table 1, CrackSQL outperforms the baselines in all the scenarios. For instance, CrackSQL achieves 3.22%-21.95% result accuracy improvement than SQLGlot, 3.22%-39.69% than jOOQ, 9.75%-64.51% than SQLines, 7.69%-34.93% than GPT-4o. The reasons are two-fold.

First, GPT-4o achieves relatively good performance in scenarios like translating among PostgreSQL and Oracle (53.85%-84.62%), but easily makes mistakes and has extremely low accuracy in some scenarios (e.g., 10.98% Acc_{RES} for MySQL→Oracle). As shown in Figure 7 (c), GPT-4o can hardly translate queries with relatively complex syntax structures, which take up 64.21% translation failures in MySQL→Oracle. Instead, CrackSQL processes the queries before translation and retains only the necessary syntax elements for LLM to translate with minimal changes required.

[9]The evaluation set and the translation result have been released at *https://github.com/weAIDB/CrackSQL*.

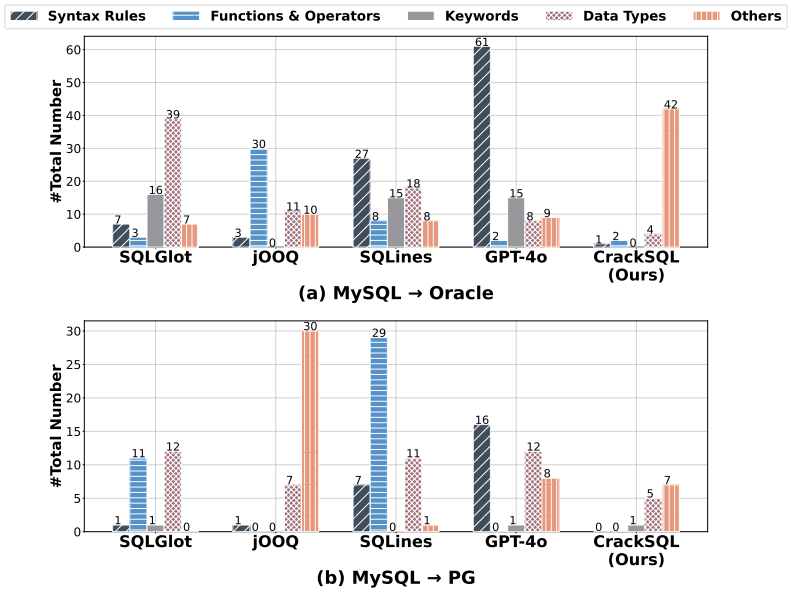

Fig. 7. Translation Error Distributions.

Besides, CrackSQL achieves much more robust performance by resolving the potential errors in intermediate steps (e.g., *Cross-Dialect Syntax Matching* and *Local-To-Global Translation*) through its hybrid validation mechanism. For instance, CrackSQL incorrectly translates TO_TIMESTAMP in PostgreSQL to UNIX_TIMESTAMP, based on the retrieved specification in MySQL. However, semantic validation identifies this as the opposite operation of the correct FROM_UNIXTIME function.

Second, rule-based translation tools not only perform worse than CrackSQL, but demonstrate less stability in different dialect translation scenarios. For example, SQLGlot behaves well in translating among MySQL and PostgreSQL, i.e., performing worse than CrackSQL but with 9.68%-15.88% higher Acc_{RES} than GPT-4o. However, SQLGlot is nearly the worst in translating among PostgreSQL and Oracle. The unstable performance is caused by the different rule complexity and development levels. For instance, we can find some mistakes in the built-in rules of SQLGlot: It converts the function JSON_EXTRACT_PATH() in PostgreSQL into the not-existing function JSON_EXTRACT() in Oracle. Instead, CrackSQL can flexibly translate queries based on the matched syntax (in any target database dialect with specifications) using the cross-dialect syntax embedding model, In this way, it can automatically accumulate new translations with minor human intervention.

7.4 Finer-Grained Analysis

Furthermore, we conduct finer-grained analysis in different metrics (see Table 1) and error distributions (see Figure 7). Table 3 showcases the valuable translation examples that are supported by CrackSQL but not adequately handled by these translation tools.

Metric Discrepancy. We further investigate the discrepancy between Acc_{EX} and Acc_{RES}, i.e., *why the syntactically correct queries produce inconsistent results*. We find that most of these cases are attributed to the inner difference of the underlying databases. For example, the numeric results are returned with different digit numbers due to the precision of the implemented data types (e.g., DECIMAL for PostgreSQL and DOUBLE for MySQL), and the same result sets are displayed in

Table 2. Ablation Study of Three Main Components.

Component		Oracle → PG		Oracle → MySQL	
Functionality-based Query Processing		Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
Query Segmentation	**Query Simplification**				
×	×	76.92	30.77	46.67	33.33
×	✓	76.92	46.15	53.33	33.33
✓	×	84.62	61.54	73.33	60.0
Cross-Dialect Embedding Model		Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
×		84.62	53.85	57.14	28.57
Local-To-Global Dialect Translation		Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
Syntactic Validation	**Semantic Validation**				
×	×	53.85	46.15	57.14	28.57
×	✓	61.54	61.54	57.14	42.86
✓	×	84.62	53.85	86.67	66.67
CrackSQL (Ours)		**92.31**	**61.54**	**93.33**	**80.0**

different orders due to the absence of the ORDER BY clause. In these cases, CrackSQL is useful to iteratively explore the potential translations (i.e., local operations to global query).

Translation Error Distribution. We analyze the translation errors according to the categories in Table 3 and identify what factors contribute to a successful translation.

• **Syntax Rules.** Although it is fundamental to ensure the translated queries strictly adhere to the syntax rules, SQLines and GPT-4o still present many syntactically incorrect queries (i.e., 35.53% for SQLines and 64.21% for GPT-4o in MySQL→Oracle). For example, SQLines fails to convert the double quotes to backticks for columns or tables (PG → MySQL) and remove AS for their alias (PG → Oracle). In contrast, CrackSQL processes queries to keep the unnecessary syntax elements unchanged. And the validation mechanism in CrackSQL ensures the translated queries strictly adhere to the target syntax rules.

• **Keywords.** As shown in Figure 7, translation tools generally make fewer mistakes in keyword handling, e.g., with error rates below 20% for jOOQ and SQLines. However, some implicit keywords still require attention. For example, in MySQL, query "ORDER BY col ASC" ranks null values first by default, but when translating to PostgreSQL, NULLS FIRST keyword must be explicitly added.

• **Functions & Operators.** Real queries involve customized functions and operators that are not supported by other databases. As shown in Figure 7, the built-in function is the primary translation error type (i.e., taking over 55.56% in the three scenarios) of jOOQ. These functions either remain unchanged or trigger a parsing error during the translation by jOOQ. For example, jOOQ cannot translate the combined operation of EXTRACT and AGE function in PostgreSQL to TIMESTAMPDIFF function in MySQL. Instead, CrackSQL normalizes customized functions into common ones, which can then be more easily translated with the retrieved function specifications and LLM knowledge.

• **Data Types.** As shown in Figure 7, data types occupy a large proportion in the translation errors of SQLGlot (e.g., 48% for MySQL→PostgreSQL). The data type errors are mainly caused by (1) inconsistent function names and (2) incorrect input data types for the functions. For example, since SQLGlot is a no-dependency SQL parser, it fails to capture the data types of columns or expressions

Table 3. Translations Errors Fixed by CrackSQL (S, K, F, D denotes translation type of Syntax Rule, Keyword, Function & Operator and Data Type respectively. Expressions are abstracted, where only the keywords and functions are highlighted in bold).

Type	Source	Target	SQLGlot	jOOQ	SQLines	GPT-4o	CrackSQL	Dialect
S1	1 / col	1 / **NULLIF**(col, 0)	✓	×	×	×	✓	MySQL → PG
S2	tbl **AS** tbl_alias	tbl tbl_alias	✓	✓	×	✓	✓	PG → Oracle
S3	**FROM** sub_query	**FROM** sub_query **AS** query_alias	×	✓	✓	×	✓	Oracle → MySQL
S4	**HAVING** ... **GROUP BY** ...	**GROUP BY** ... **HAVING** ...	✓	✓	×	✓	✓	Oracle → PG
K1	**GROUP BY ROLLUP**(col_list)	**GROUP BY** col_list **WITH ROLLUP**	×	✓	×	×	✓	PG → MySQL
K2	... tbl1 **FULL OUTER JOIN** tbl2 **ON** ...	... tbl1 **LEFT OUTER JOIN** tbl2 **ON** ... **UNION ALL** ... tbl1 **RIGHT OUTER JOIN** tbl2 **ON** ... **WHERE NOT EXISTS** (**SELECT** 1 **FROM** tbl1 **WHERE** ...)	✓	×	×	×	✓	PG → MySQL
K3	**LIMIT** num1, num2	**LIMIT** num2 **OFFSET** num1	✓	✓	×	✓	✓	MySQL → PG
K4	**WHERE ROWNUM** <= num	**LIMIT** num	×	✓	×	×	✓	Oracle → PG
K5	**FROM** DUAL	/	×	✓	✓	×	✓	Oracle → PG
F1	**EXTRACT**(unit **FROM AGE**(time1, time2))	**TIMESTAMPDIFF**(unit, time2, time1)	×	×	×	×	✓	PG → MySQL
F2	str1 **ILIKE** str2	**LOWER**(str1) **LIKE LOWER**(str2)	✓	✓	×	×	✓	PG → MySQL
F3	**TO_TIMESTAMP**(unix_time)	**FROM_UNIXTIME**(unix_time)	✓	×	×	✓	✓	PG → MySQL
F4	**CURDATE**()	**CURRENT_DATE**	×	✓	×	✓	✓	MySQL → PG
F5	**LAST_DAY**(time)	**DATE_TRUNC**('MONTH', time) + **INTERVAL** '1 MONTH' - **INTERVAL** '1 DAY'	✓	×	×	✓	✓	MySQL → PG
D1	**CAST**(col **AS bytea**)	**CAST**(col **AS BINARY**)	×	×	×	✓	✓	PG → MySQL
D2	**DATE_FORMAT**(time1, unit_format) - **DATE_FORMAT**(time2, unit_format)	**EXTRACT**(unit **FROM** time1) - **EXTRACT**(unit **FROM** time2)	×	×	×	×	✓	MySQL → PG
D3	date_obj = time	date_obj = **TO_DATE**(time, format)	×	×	×	✓	✓	MySQL → Oracle
D4	**DATETIME**	**TIMESTAMP**	×	✓	✓	✓	✓	MySQL → Oracle
D5	**SUM**(bool_expr)	**SUM**(**CASE WHEN** bool_expr **THEN** 1 **ELSE** 0 **END**)	×	✓	✓	✓	✓	MySQL → Oracle

and conduct proper type conversion. In contrast, CrackSQL retrieves specifications that clarify the input data types. And CrackSQL can utilize the type information in our BNF tree to check the validity of each component in the syntax elements.

7.5 Ablation Study

We conduct ablation studies to verify the effectiveness of the main components in CrackSQL, including 7 variants over the three main components. The results in Table 2 demonstrate that all the designs in these components contribute to the effectiveness of CrackSQL.

7.5.1 Functionality-based Processing

We investigate the effectiveness of the *Functionality-based Query Processing* component by assessing the translation accuracy of three variants, including (1) CrackSQL w/o *Query Segmentation*, (2) CrackSQL w/o *Query Simplification*, and (3) CrackSQL w/o *Query Segmentation* and w/o *Query Simplification*. The results are shown in Table 2, and we have the following observations.

First, *Query Segmentation* significantly enhances translation accuracy (by 48.91% on average). It breaks down the lengthy query into multiple succinct segments, allowing LLM to focus on translating each segment independently while ignoring irrelevant parts (e.g., translating the DATE_FORMAT function within a complex MySQL query). Second, *Query Simplification* also improves translation accuracy (by 11.74% on average). It simplifies customized functions into common ones for easier function matching (e.g., replacing the unique ILIKE function in PostgreSQL with common LOWER and LIKE functions) and converts indispensable parts within segments into simplified formats, thereby reducing translation complexity (e.g., abstracting the subquery as `table_id`).

7.5.2 Cross-Dialect Embedding Model

We first verify the effectiveness of *Cross-Dialect Embedding Model*. As shown in Table 4, CrackSQL with the embedding model achieves 45.95% higher translation accuracy on average. This improvement can be attributed to the translation knowledge that LLM lacks but is retrieved by the embedding model, including the usage of equivalent keywords and built-in functions. For instance, the TO_TIMESTAMP function in PostgreSQL is equivalent to the FROM_UNIXTIME function in

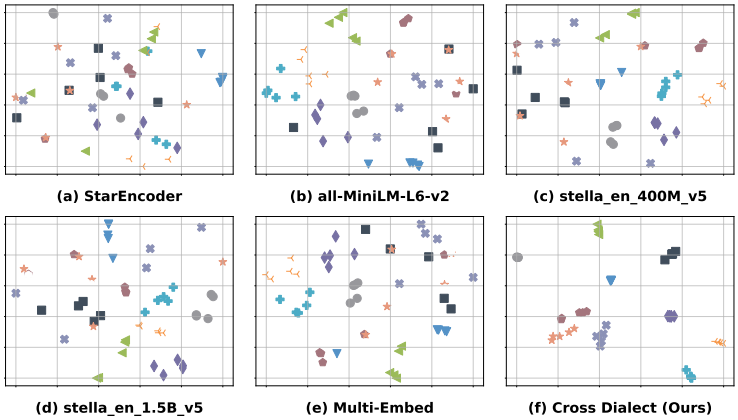

Fig. 8. t-SNE of Embedding Models (Same markers denotes functionally equivalent specifications).

Table 4. Precision (%) of Different Embedding Models.

Model	PG → MySQL			MySQL → PG			Oracle → PG		
	k=1	k=3	k=5	k=1	k=3	k=5	k=1	k=3	k=5
BM25	35.71	46.94	55.61	53.92	66.67	71.57	4.29	15.71	51.43
StarEncoder	11.73	18.88	23.47	1.96	2.94	2.94	0.0	0.0	0.0
all-MiniLM-L6-v2	38.27	54.08	61.73	43.14	69.61	75.49	24.29	41.43	51.43
stella_en_400M_v5	43.88	68.37	71.43	50.98	79.41	83.33	53.66	76.83	84.15
stella_en_1.5B_v5	39.29	60.20	64.29	34.31	48.04	53.92	19.51	28.05	31.71
MultiEmbed	45.92	60.71	67.35	50.98	74.51	85.29	30.0	51.43	61.43
Cross Dialect (Ours)	**74.49**	**89.29**	**95.41**	**76.47**	**90.20**	**94.12**	**61.43**	**85.71**	**91.43**

MySQL, which have different names but the same functionality. By leveraging this knowledge, LLMs can be guided to perform more accurate translations.

We then compare *Cross-Dialect Embedding Model* with the following baselines for a more fine-grained analysis. (1) `BM25`: a lexical-based keyword search method that ranks the specifications based on the query terms in the set of all the documents; (2) `StarEncoder`: a transformer-based model [28] trained on 80+ programming languages (including SQL); (3) `all-MiniLM-L6-v2`: the default embedding model utilized in a LLM system framework (i.e., the langchain project); (4) `stella_en_400M_v5`: the top-1 embedding model in the MTEB leaderboard whose parameters volume is less than 1B; (5) `stella_en_1.5B_v5`: a larger version of stella with 1.5B parameters; (6) `MultiEmbed`: directly concatenate the embeddings generated by the above model. We evaluate the precision of the retrieved top-k specifications (i.e., from 1 to 5) by performing a similarity search with the embeddings generated by these models.

Retrieval Precision. As shown in Table 4, our embedding model obtains the highest precision values when k varies from 1 to 5, which is 86.34% higher than others on average. The code-only model (`StarEncoder`) cannot retrieve the equivalent specifications and causes low precision (lower than 20%). It indicates the effectiveness of the fusion of multiple embeddings in our model, where each query specification is associated with the syntax tree elements and textual illustrations. Thus, the embeddings considering both syntax tree and specification factors help to achieve more accurate equivalent syntax matching.

Table 5. Translation Accuracy (%) v.s. Underlying LLM.

Method	Oracle → PG		Oracle → MySQL	
	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
CodeLlama-7B	7.69	7.69	0.0	0.0
CrackSQL (CodeLlama-7B)	**38.46**	**30.77**	**50.0**	**42.86**
Llama3.1-8B	76.92	46.15	73.33	53.33
CrackSQL (Llama3.1-8B)	**92.31**	**61.54**	**93.33**	**73.33**
GPT-4o	84.62	53.85	80.0	73.33
CrackSQL (GPT-4o)	**92.31**	**61.54**	**93.33**	**80.0**

Table 6. Translation Accuracy (%) with Finetuning (SFT-SELF: Training over MySQL → Oracle data samples; SFT-BRIDGE: Training over MySQL → PG and PG → Oracle data samples).

Method	MySQL → Oracle		MySQL → PG	
	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
CodeLlama-7B	17.07	7.32	17.46	17.46
CodeLlama-7B (SFT-SELF)	29.27	20.73	44.44	42.86
CodeLlama-7B (SFT-BRIDGE)	43.9	34.18	52.46	45.9
CrackSQL (CodeLlama-7B)	**48.78**	**36.59**	**60.32**	**57.14**
Llama3.1-8B	17.07	17.07	58.73	50.79
Llama3.1-8B (SFT-SELF)	29.27	20.73	33.33	33.33
Llama3.1-8B (SFT-BRIDGE)	29.27	20.73	42.86	42.86
CrackSQL (Llama3.1-8B)	**53.16**	**36.59**	**72.88**	**54.24**

Functionality Embeddings. We further investigate the embeddings generated by different models by projecting them into the 2-D space with t-SNE [41]. As shown in Figure 8, the embeddings generated by our model can cluster the specifications by their functionalities better than other methods. It validates the effectiveness of the proposed retrieval-enhanced contrastive learning. Our contrastive learning bridges the gap between different dialects by leveraging a series of dialect-specific query specification samples to pulls in the functionally equivalent ones while pushes apart the others among the dialects. In contrast, existing embedding models typically only consider the correlation within one dialect independently and do not exclusively attempt to identify the connections among dialects.

7.5.3 Local-To-Local Translation

We explore effectiveness of CrackSQL variants by changing two validation mechanisms (i.e., *Syntactic Validation* and *Semantic Validation*) separately for syntactic and semantic checking in *Local-To-Global Translation*. As shown in Table 2, we notice that: (1) *Syntactic Validation* improves the translation accuracy, especially for the executable ratio (Acc_{EX}), by 57.13% on average. It locates and identifies the segments that violate the dialect standards and provides LLM with explicit feedback to handle these segments (e.g., function CURDATE() does not exist in PostgreSQL); (2) *Semantic Validation* helps to achieve a 20.12% higher translation accuracy, especially for the result consistency (Acc_{RES}), on average. Because it complements and figures out the subtle semantic differences not detected by grammar checking in syntactic validation. For instance, MySQL functions CURDATE and CURRENT_TIMESTAMP return current time with different units and formats.

Table 7. Accuracy (%) with Execution Feedback.

Method	PG → MySQL		MySQL → PG	
	Acc_{EX}	Acc_{RES}	Acc_{EX}	Acc_{RES}
Llama3.1-8B (w/o feedback)	61.29	35.48	68.25	58.73
Llama3.1-8B (w/ feedback)	67.74	45.16	69.84	60.32
GPT-4o (w/o feedback)	61.29	61.29	50.79	44.44
GPT-4o (w/ feedback)	64.52	51.61	76.19	68.25
CrackSQL (Ours)	**87.1**	**74.19**	**85.71**	**79.37**

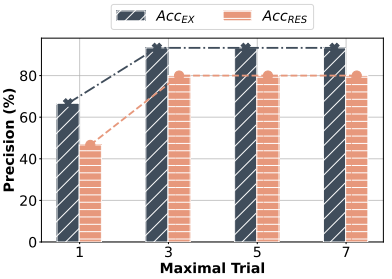

Fig. 9. Accuracy (%) v.s. Maximum Trials.

7.5.4 LLM Settings

We validate the effectiveness of CrackSQL with different LLM settings, including (*i*) various LLM types, (*ii*) with / without the execution feedback, and (*iii*) the maximum trial numbers.

LLM Types. We test two localized LLMs: (1) `CodeLlama-7B`: A Llama2 model finetuned for code synthesis and understanding [1]; (2) `Llama3.1-8B-Instruct`: A general-purpose Llama3 model optimized for multilingual cases [4].

As shown in Table 5, CrackSQL obviously improves LLM'ts dialect translation performance, i.e., achieving 6.67%-13.33% accuracy improvement for GPT-4o, 15.39%-20% for `Llama3.1-8B-Instruct`, and 23.08%-50% for `CodeLlama-7B`. The reasons are two-fold. First, with techniques like query processing and cross-dialect syntax matching, CrackSQL reduces the dependency on the LLM by using it only to ensure target syntax adherence during the transformation of partial syntax elements. Second, CrackSQL (`Llama3.1-8B-Instruct`) achieves performance similar to CrackSQL (GPT-4o), with only 6.67% worse in Acc_{RES} of Oracle→MySQL. This verifies CrackSQL is robust across different LLMs and does not heavily depend on the advanced skills like reasoning, making it a versatile solution for dialect translation.

LLM Finetuning. We also evaluate the impact of supervised fine-tuning (LoRA) on two datasets with SQLs translated by SQLGlot. (1) `SFT-SELF`: a training dataset with 528 distinct MySQL→Oracle samples, and (2) `SFT-BRIDGE`: a training dataset with 864 distinct MySQL→PG and PG→Oracle samples. None of the samples in the evaluation set is included in the training datasets.

As shown in Table 6, CrackSQL obtains a stable 38.91% higher accuracy over all the dialect pairs on average. Instead, the fine-tuning methods are less stable. For example, the training of `SFT-BRIDGE` over other data samples exhibits higher translation accuracy than exclusive training of `SFT-SELF` over the MySQL→Oracle data samples. Moreover, Llama3.1-8B after fine-tuning (i.e., 38.10% accuracy on average) even performs worse over MySQL→PG data samples than the original version (i.e., 54.76% accuracy on average). Because fine-tuning can negatively affect the model's generalizability [21], leading to errors in translations that were previously handled correctly (e.g., mistakenly assume the MySQL data type DATETIME also exists in PostgreSQL).

Execution Feedback. We investigate the impact of providing LLM with the execution feedback (i.e., execution errors of the translated query) on the translation accuracy. As displayed in Table 7, LLMs with feedback from the execution environment still have problems in accurate dialect translation, where simply offering the execution feedback to LLM still cannot make all the queries executable (i.e., Acc_{EX} is lower than 100%). It can be attributed to (1) failing to accurately locate the target segments with the execution feedback, and (2) missing critical dialect knowledge (e.g., the correct usage of function arguments in the target dialect). In contrast, CrackSQL effectively locates the necessary segments through *Functionality-based Query Processing* and *Cross-Dialect Syntax Matching*, and enables robust translation under the hybrid validation mechanism.

Trial Numbers. We investigate the impact of different maximum trial numbers (ranging from 1 to 7) for LLM translation. As shown in Figure 9 , the translation accuracy of CrackSQL converges when

the maximal trial arrives 3, which is a suitable hyper-parameter value to balance the translation performance and cost. Besides, it also reflects CrackSQL can accurately translate most queries within limited trials, where additional attempts are unnecessary once the incompatible operations are accurately translated.

8 Related Work

Dialect Translation Engines. Existing dialect translation engines [3, 7, 8] integrate limited translation rules maintained by humans and cannot translate in many cases (see Table 3).

Logical Query Rewrite. *(1) Rewrite Rule Discovery:* Existing methods [31, 38, 39, 43] identify rewrite rules that can be integrated in engines like Calcite [15]. For instance, the most recent work [43] identifies rules by heuristically searching for equivalent queries in single database dialect. *These rewrite rules cannot be reused in dialect translation and do not involve cross-dialect syntax alignment. (2) Rewrite Rule Order Selection:* Tools like ARE-SQL [17] and SOAR [6] recommend appropriate rules by database statistics. Further, the method [47] explores beneficial rewrite orders using a tree search-based strategy. Instead, *dialect translation is rule order insensitive.*

LLM for Databases. *(1) LLM for rule discovery:* The vision paper [36] utilizes LLMs to generate translation rules in text rather than formalized code. There lack necessary technical details (e.g., how to use the textual rules) and experiment analysis. Similarly, the work [29] utilizes course learning and RAG techniques to improve LLMs' ability to determine optimal sequences for applying rewrite rules. *(1) LLM for task-solving:* Works like [30, 40, 48] prompt or finetune LLMs to fulfill database tasks. For database diagnosis, the works [40, 48] utilize (multiple) LLMs to analyze abnormal metrics and activities and report root causes. They are augmented with diagnosis expertise like documents. For query rewrite, the work [30] leverages LLM to rewrite queries via iterative self-reflection.

9 Conclusion

Automatic dialect translation is essential in many real applications. In this paper, we introduced a query processing method that segments and simplifies input queries to facilitate translation. We developed *Cross-Dialect Embedding Model*, a model that computes functionality embeddings based on syntax elements and specifications to effectively match syntax variants. We proposed the *Local-to-Global Translation Strategy*, which enhances translation accuracy by constraining target-syntax-enhanced LLM during the translation of limited query operations. Experimental results demonstrated the effectiveness of our approach over baseline methods.

Despite its effectiveness, CrackSQL has three main limitations. First, the translation types supported by CrackSQL are constrained by the available BNF grammar. We plan to explore automatic grammar integration methods [35] to accommodate a broader range of translations (e.g., for UDFs and stored procedures). Second, while CrackSQL achieves significantly higher translation accuracy than the baselines, its accuracy could be further enhanced (e.g., resolving the potential precision loss issues like DOUBLE v.s. DOUBLE PRECISION). Third, CrackSQL requires human intervention when dealing with new dialects that involve incomplete or poorly structured documents. For instance, some dialect documents may only list function names or combine multiple specifications within single paragraphs, necessitating manual annotation and clarification.

Acknowledgments

Wei Zhou and Yuyang Gao are co-first authors. Xuanhe Zhou and Guoliang Li are corresponding authors. This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62232009, 62102215), Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] Code-Llama. *(Model)*. https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf Last accessed on 2024-10.
[2] GPT-4o. *(Model)*. https://openai.com/index/hello-gpt-4o/ Last accessed on 2024-10.
[3] jOOQ. *(Tool)*. https://www.jooq.org/ Last accessed on 2024-10.
[4] Llama3.1. *(Model)*. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct Last accessed on 2024-10.
[5] Ora2PG. *(Tool)*. https://ora2pg.darold.net/ Last accessed on 2024-10.
[6] SOAR. *(Tool)*. https://github.com/xiaomi/soar/ Last accessed on 2024-10.
[7] SQLGlot. *(Tool)*. https://sqlglot.com/sqlglot.html Last accessed on 2024-10.
[8] SQLines. *(Tool)*. https://www.sqlines.com/ Last accessed on 2024-10.
[9] StackExchange. *(Website)*. https://archive.org/details/stackexchange Last accessed on 2024-10.
[10] TPC-DS Benchmark. *(TPC)*. https://www.tpc.org/tpcds
[11] TPC-H Benchmark. *(TPC)*. https://www.tpc.org/tpch
[12] Saleh Albelwi. 2022. Survey on Self-Supervised Learning: Auxiliary Pretext Tasks and Contrastive Learning Methods in Imaging. *Entropy* 24, 4 (2022), 551. doi:10.3390/E24040551
[13] Chitra Babu and G Gunasingh. 2016. DESH: Database evaluation system with hibernate ORM framework. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2549–2556.
[14] Tao Bai, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. 2021. Recent Advances in Adversarial Training for Adversarial Robustness. In *IJCAI*. 4312–4321.
[15] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 221–230. doi:10.1145/3183713.3190662
[16] Hongliu Cao. 2024. Recent advances in text embedding: A Comprehensive Review of Top-Performing Methods on the MTEB Benchmark. *arXiv preprint arXiv:2406.01607* (2024).
[17] Arlino H. M. de Araújo, José Maria Monteiro, José Antônio Fernandes de Macêdo, Júlio A. Tavares, Angelo Brayner, and Sérgio Lifschitz. 2014. On Using an Online, Automatic and Non-Intrusive Approach for Rewriting SQL Queries. *J. Inf. Data Manag.* 5, 1 (2014), 28–39. https://sol.sbc.org.br/journals/index.php/jidm/article/view/1517
[18] Franklin L. Deremer. 1969. Generating parsers for BNF grammars. In *AFIPS Spring Joint Computing Conference (AFIPS Conference Proceedings, Vol. 34)*. 793–799.
[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.
[20] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-situ cross-database query processing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2794–2807.
[21] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. *CoRR* abs/2403.14608 (2024).
[22] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proc. VLDB Endow.* 13, 10 (2020), 1737–1750.
[23] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (2024), 1939–1952.
[24] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.* 17, 11 (2024), 3318–3331.
[25] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *Proc. VLDB Endow.* 17, 12 (2024), 4213–4216.
[26] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *NeurIPS*.
[27] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. http://papers.nips.cc/paper_files/paper/2023/hash/83fc8fab1710363050bbd1d4b8cc0021-Abstract-Datasets_and_Benchmarks.html
[28] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade,

Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *arXiv Preprint* (2023). https://arxiv.org/abs/2305.06161

[29] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *CoRR* abs/2404.12872 (2024).

[30] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. *CoRR* abs/2403.09060 (2024). doi:10.48550/ARXIV.2403.09060 arXiv:2403.09060

[31] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *SIGMOD*, Haran Boral and Per-Åke Larson (Eds.). ACM Press, 18–27. doi:10.1145/50202.50204

[32] Edward Ma. 2019. NLP Augmentation. https://github.com/makcedward/nlpaug.

[33] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *arXiv preprint arXiv:2402.06196* (2024).

[34] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022. MTEB: Massive Text Embedding Benchmark. *arXiv preprint arXiv:2210.07316* (2022). doi:10.48550/ARXIV.2210.07316

[35] Hannes Mühleisen and Mark Raasveldt. 2025. Runtime-Extensible Parsers. *CIDR* (2025).

[36] Amadou Latyr Ngom and Tim Kraska. 2024. Mallet: SQL Dialect Translation with LLM Rule Generation. In *aiDM 2024*. ACM, 3:1–3:5. doi:10.1145/3663742.3663973

[37] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *ICSE*. ACM, 82:1–82:13.

[38] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, Michael Stonebraker (Ed.). ACM Press, 39–48. doi:10.1145/130283.130294

[39] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *SIGMOD*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 435–446. doi:10.1145/233269.233360

[40] Vikramank Y. Singh, Kapil Vaidya, Vinayshekhar Bannihatti Kumar, Sopan Khosla, Balakrishnan Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2024. Panda: Performance Debugging for Databases using LLM Agents. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p6-singh.pdf

[41] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[42] Shenzhi Wang, Yaowei Zheng, Guoyin Wang, Shiji Song, and Gao Huang. 2024. Llama3.1-8B-Chinese-Chat. doi:10.57967/hf/2779

[43] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD Conference*. 94–107.

[44] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. CrackSQL: A Hybrid SQL Dialect Translation System Powered by Large Language Models. *arXiv Preprint* (2025). https://arxiv.org/abs/2504.00882

[45] Wei Zhou, Chen Lin, Xuanhe Zhou, and Guoliang Li. 2024. Breaking It Down: An In-depth Study of Index Advisors. *Proc. VLDB Endow.* 17, 10 (2024), 2405–2418.

[46] Wei Zhou, Chen Lin, Xuanhe Zhou, Guoliang Li, and Tianqing Wang. 2024. TRAP: Tailored Robustness Assessement for Index Advisors via Adversarial Perturbation. In *ICDE*. to appear.

[47] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.

[48] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2024. D-Bot: Database Diagnosis System using Large Language Models. *Proc. VLDB Endow.* 17, 10 (2024), 2514–2527. https://www.vldb.org/pvldb/vol17/p2514-li.pdf

Received October 2024; revised January 2025; accepted February 2025