

# QTRAN: Extending Metamorphic-Oracle Based Logical Bug Detection Techniques for Multiple-DBMS Dialect Support

LI LIN, School of Informatics, Xiamen University, China

QINGLIN ZHU, School of Informatics, Xiamen University, China

HONGQIAO CHEN, School of Informatics, Xiamen University, China

ZHUANGDA WANG, School of Informatics, Xiamen University, China

RONGXIN WU, School of Informatics, Xiamen University, China

XIAOHENG XIE, Ant Group, China

Metamorphic testing is a widely used method to detect logical bugs in Database Management Systems (DBMSs), referred to herein as **MOLT** (Metamorphic-Oracle based Logical Bug Detection Technique). This technique involves constructing SQL statement pairs, including original and mutated queries, and assessing whether the execution results conform to predefined metamorphic relations to detect logical bugs. However, current MOLTs rely heavily on specific DBMS grammar to generate valid SQL statement pairs, which makes it challenging to adapt these techniques to various DBMSs with different grammatical structures. As a result, only a few popular DBMSs, such as PostgreSQL, MySQL, and MariaDB, are supported by existing MOLTs, with extensive manual effort required to expand to other DBMSs. Given that many DBMSs remain inadequately tested, there is a pressing need for a method that enables effortless extension of MOLTs across diverse DBMSs.

In this paper, we propose QTRAN, a novel LLM-powered approach that automatically extends existing MOLTs to various DBMSs. Our key insight is to translate SQL statement pairs to target DBMSs for metamorphic testing from existing MOLTs using LLMs. To address the challenges of LLMs' limited understanding of dialect differences and metamorphic mechanisms, we propose a two-phase approach comprising the transfer and mutation phases. QTRAN tackles these challenges by drawing inspiration from the developer's process of creating a MOLT, which includes understanding the grammar of the target DBMS to generate original queries and employing a mutator for customized mutations. The transfer phase is designed to identify potential dialects and leverage information from SQL documents to enhance query retrieval, enabling LLMs to translate original queries across different DBMSs accurately. During the mutation phase, we gather SQL statement pairs from existing MOLTs to fine-tune the pretrained model, tailoring it specifically for mutation tasks. Then we employ the customized LLM to mutate the translated original queries, preserving the defined relationships necessary for metamorphic testing.

We implement our approach as a tool and apply it to extend four state-of-the-art MOLTs for eight DBMSs: MySQL, MariaDB, TiDB, PostgreSQL, SQLite, MonetDB, DuckDB, and ClickHouse. The evaluation results show that over 99% of the SQL statement pairs transferred by QTRAN satisfy the metamorphic relations required for testing. Furthermore, we have detected 24 logical bugs among these DBMSs, with 16 confirmed as unique and previously unknown bugs. We believe that the generality of QTRAN can significantly enhance the reliability of DBMSs.

CCS Concepts: • **Security and privacy** → *Database and storage security*.

Authors' Contact Information: [Li Lin](#), School of Informatics, Xiamen University, Xiamen, China, [linli1210@stu.xmu.edu.cn](mailto:linli1210@stu.xmu.edu.cn); [Qinglin Zhu](#), School of Informatics, Xiamen University, Xiamen, China, [23020241154481@stu.xmu.edu.cn](mailto:23020241154481@stu.xmu.edu.cn); [Hongqiao Chen](#), School of Informatics, Xiamen University, Xiamen, China, [23020241154371@stu.xmu.edu.cn](mailto:23020241154371@stu.xmu.edu.cn); [Zhuangda Wang](#), School of Informatics, Xiamen University, Xiamen, China, [wangzhuangda@stu.xmu.edu.cn](mailto:wangzhuangda@stu.xmu.edu.cn); [Rongxin Wu](#), School of Informatics, Xiamen University, Xiamen, China, [wurongxin@xmu.edu.cn](mailto:wurongxin@xmu.edu.cn); [Xiaoheng Xie](#), Ant Group, ShenZhen, China, [xiexie@antgroup.com](mailto:xiexie@antgroup.com).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2994-970X/2025/7-ARTISSTA033

<https://doi.org/10.1145/3728908>

Additional Key Words and Phrases: Database Testing, Metamorphic Testing, SQL Dialects

### ACM Reference Format:

Li Lin, Qinglin Zhu, Hongqiao Chen, Zhuangda Wang, Rongxin Wu, and Xiaoheng Xie. 2025. QTRAN: Extending Metamorphic-Oracle Based Logical Bug Detection Techniques for Multiple-DBMS Dialect Support. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA033 (July 2025), 22 pages. <https://doi.org/10.1145/3728908>

## 1 Introduction

Database Management Systems (DBMSs) are crucial in applications such as online banking and e-commerce [19, 52]. However, the complex code logic and diverse functionalities of DBMSs often make them susceptible to bugs, especially logical bugs that result in incorrect result sets being returned without obvious symptoms [8, 20, 24, 26, 37–39]. Automated testing approaches use test oracles [21] to verify the correctness of DBMS results. Notably, metamorphic oracle, a widely-used method that constructs SQL statements maintaining either exact [25, 28, 29, 37–39, 44] or approximate equivalence [20] with the original query, has been recognized as state-of-the-art in DBMS testing for logical bug detection [20, 38, 40]. The detection of logical bugs is facilitated by comparing the results of these mutated queries with those of the original queries. Unlike traditional fuzzing [1, 24], it is essential to ensure both syntactic and semantic correctness, while strictly preserving the metamorphic relationship between the queries. We call this approach **MOLT** (Metamorphic-Oracle based Logical Bug Detection Technique).

To generate SQL statement pairs suitable for MOLTs, including *original queries* and *mutated queries*, a general idea is to model the SQL grammar as an Abstract Syntax Tree (AST). This AST model forms the backbone for generating original queries, ensuring syntactic correctness based on the specific grammar of the target DBMS. These statements are then mutated according to specific patterns to ensure that the mutated queries strictly adhere to a defined metamorphic relationship with the original queries. For instance, Pinolo [20] customizes its parser specifically for MySQL syntax. It randomly generates an original query based on the AST in the form of “SELECT (COLLATION(`f`)) from t” and then, following a specific pattern, mutates this query into “SELECT DISTINCT (COLLATION(`f`)) from t”, ensuring that the results of the original query subsume the results of the mutated query.

Currently, state-of-the-art MOLTs predominantly depend on the grammar of specific DBMSs to generate valid SQL statement pairs. However, this approach requires extensive manual effort, limiting current MOLTs to support only a few popular DBMSs, such as PostgreSQL, MySQL, and MariaDB. For instance, to integrate PostgreSQL’s grammar, SQLRight [29] required an additional 93,671 lines of code. Despite the existence of 423 different DBMSs in the market today [43], most of which utilize unique grammar, there remains a pressing need to enhance their security. Consequently, there is a great need for a method that can easily extend existing MOLTs to diverse database environments with minimal effort.

In recent years, there has been a vast proliferation of software engineering applications built upon large language models (LLMs) [16, 41, 46], from which we observe the exceptional performance of LLMs in SQL generation [27, 51]. A potential solution [48] is to treat LLMs as domain experts for multiple DBMS SQL transfer. By crafting suitable prompts, we can facilitate the direct transfer of SQL statement pairs from the original database to the target database. This eliminates the need for an AST model due to grammar differences. However, the direct transfer of SQL statement pairs across DBMSs using LLMs poses two significant challenges:

**Challenge#1. Limited understanding of the dialect differences between DBMSs.** LLMs lack a deep understanding of the dialect differences between DBMSs, which can lead to numerous syntactic and semantic errors [23], potentially resulting in functionally inequivalent queries. For example, when transferring a MySQL query “SELECT (COLLATION(`f`)) from t” to PostgreSQL,

an LLM lacking domain-specific knowledge might incorrectly simplify it to “SELECT f from t”, omitting the dialect-specific functions for character set comparison and sorting. This leads to a functionally inequivalent transfer because the LLM fails to recognize that in PostgreSQL, character set comparison and sorting can be performed using the COLLATE FOR function. In order to improve the transfer of semantically equivalent queries, it is highly desirable to enhance the LLM’s understanding of DBMS-specific dialects.

**Challenge#2. Limited understanding of the metamorphic mechanisms.** Secondly, the limited understanding of metamorphic mechanisms by LLMs compromises their ability to maintain the defined metamorphic relationship between original and mutated queries [23]. Without this relationship, it becomes difficult to compare query results effectively, hindering the detection of logical bugs. For example, as shown in Figure 1, when both the original and mutated MySQL queries are translated into PostgreSQL, the mutated query loses the “DISTINCT” keyword, resulting in a failure to maintain the predefined under-approximation relationship between the two queries. Without this defined relationship, the ability to detect bugs through comparison of query results is lost, severely undermining the effectiveness of metamorphic testing in finding logical bugs.

To address the challenges, we introduce QTRAN, a novel LLM-powered approach that automatically extends existing MOLTs to various DBMSs. Our inspiration derives from observing how developers create MOLTs in practice. To enhance the transfer capabilities of LLMs, our approach mines dialect knowledge from SQL documents and extracts metamorphic relationships and mechanisms from existing MOLTs’ SQL statement pairs. Our method includes two key phases: transfer and mutation, each designed to overcome the aforementioned challenges:

① **Transfer Phase:** In this phase, we aim to identify potential dialects and leverage information from SQL documents to improve retrieval, helping LLMs translate original queries to different DBMSs. Firstly, we construct a feature knowledge base for each DBMS to store dialect information, gather feature data from SQL documents. These features are mapped to equivalent elements in the target DBMS using a pre-established knowledge base enhanced by Retrieval-Augmented Generation (RAG). The enriched information, along with the customized prompt, is then fed into the LLM. This tailored input is designed to mitigate the risks of syntactic and semantic errors during the transfer process, ensuring that SQL queries transferred from the original database to the target database maintain their functional integrity.

② **Mutation Phase:** Post-transfer, the mutation phase employs a deep understanding of metamorphic mechanisms to mutate the original SQL queries. We gather SQL statement pairs from existing MOLTs to fine-tune the pretrained model, tailoring it specifically for mutation tasks. By learning from established metamorphic mechanisms, the mutation LLM focuses on critical transformation elements that preserve the defined relationships necessary for effective testing. A critical consideration is that although existing MOLT provides mutated queries for the original ones, directly transferring these mutated queries to the target DBMS using the transfer LLM may not be effective. The diversity and randomness of SQL dialects (e.g., PostgreSQL allows type conversion to be expressed as either “f : text” or “CAST(f AS TEXT)”) prevent the transfer LLM from ensuring consistent results, potentially disrupting the metamorphic relationship. For example, our experimental evaluations showed that, the success rate of direct transfer of existing mutated queries is less than 35%. Therefore, a dedicated mutation LLM is essential to preserve these relationships.

We implement QTRAN based on GPT-4o-mini. Compared to other conventional MOLTs which require intensive labor to write a large amount of SQL grammar adaptation code, QTRAN is more conveniently adapted to various DBMSs because it is free from the dependencies of grammar. For evaluation, we implement and apply QTRAN to extend four state-of-the-art MOLTs for eight DBMSs: MySQL, MariaDB, TiDB, PostgreSQL, SQLite, MonetDB, DuckDB, and ClickHouse. The

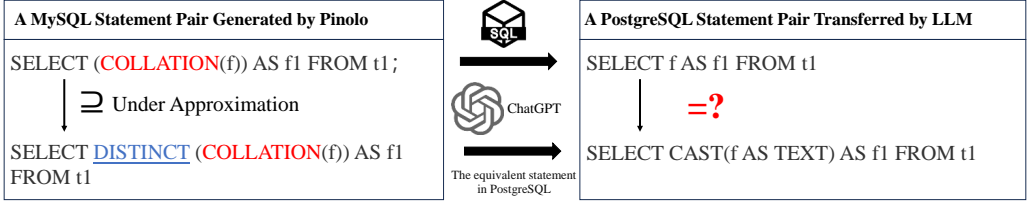


Fig. 1. **A failed example in Pinolo using LLMs to transfer SQL statement pairs directly.** This is a simplified example. Red text indicates dialect elements, highlighting how ChatGPT, due to a lack of dialect knowledge, dropped the COLLATION function during the transfer process. Blue text represents critical code elements for metamorphic testing; ChatGPT lost the DISTINCT keyword during the transfer process.

experimental results indicate that QTRAN ensures that over 99% of the transferred SQL statement pairs satisfy the metamorphic relations required for testing, demonstrating its efficacy as an ideal tool for extending MOLTs to support metamorphic testing across diverse database systems. Furthermore, we have detected 24 logical bugs among these DBMSs, of which 16 have been confirmed as unique and previously unknown bugs. We believe that the generality of QTRAN can greatly help improve the reliability of DBMSs.

In summary, we have the following contributions:

- We introduce a two-phase approach using LLMs that significantly simplifies the extension of MOLTs to new DBMSs.
- We implement the approach in QTRAN. In evaluation, QTRAN ensures that over 99% of the transferred SQL statement pairs are suitable for metamorphic testing. Additionally, QTRAN discovered 24 previously unknown logical bugs, 16 of which have been confirmed.
- We have contributed a feature knowledge base containing dialect information for eight different DBMSs, which plays a crucial role in enhancing the SQL translation task.

## 2 Background and Motivation

This section introduces basic knowledge about DBMSs such as SQL grammar and metamorphic testing, and discusses the motivation, challenges associated with extending MOLTs in various DBMSs, as well as our basic idea.

**DBMS and SQL Grammar.** A Database Management System (DBMS) is software that stores and retrieves data based on a specified database model. Structured Query Language (SQL) is a domain-specific language used to manage data within a DBMS, and SQL grammar describes both the syntax and semantics of the language. Typically, most DBMS products comply with the basic features of the ANSI SQL standard [2] for common operations but extend support for advanced functionalities through their own unique SQL dialects. However, the grammar of these SQL dialects are not compatible with one another, as similar features might be implemented with distinct grammar variations across different DBMSs. For example, while the character set comparison and sorting function is implemented as COLLATION in MySQL, it is implemented as COLLATE FOR in PostgreSQL. There are approximately 423 popular DBMS products [43], each differing in supported features, such as data types, functions, operators, etc.

**Metamorphic Testing in DBMSs.** Recent years have witnessed tremendous efforts in resolving the test oracle for logical bug detection in the DBMSs. Notably, the metamorphic testing based approach MOLT has been recognized to be state-of-the-art in DBMS testing for logical bug detection [20, 38, 40]. Metamorphic testing validates database systems by applying transformations to original queries to generate new, mutated queries. Let an original query be represented as  $Q_0$ , and

a set of transformed queries  $\{Q_1, Q_2, \dots, Q_n\}$  derived from applying metamorphic transformations. The testing process validates that for any test database  $D$ , the following relation holds:

$$D(Q_0) \approx D(Q_j), \quad \forall j \in [1, n]$$

where  $\approx$  represents either equivalence ( $=$ ) or an approximate relation based on a predefined metamorphic relationship. The key elements of metamorphic testing in this context include **SQL statement pairs**, comprising original queries  $Q_0$  and their mutated counterparts  $Q_1, Q_2, \dots$ , and the verification that their execution results uphold either equivalence or near-equivalence under the mutations applied.

The metamorphic testing process is conducted as follows: (i) Generate the original query  $Q_0$ , serving as the basis for testing. (ii) Construct mutated queries  $Q_1, Q_2, \dots, Q_n$  by applying predefined metamorphic transformations to  $Q_0$ . (iii) Execute  $Q_0$  on a test database  $D$  to obtain results  $R_0 = D(Q_0)$ . (iv) Execute each mutated query  $Q_j$  on  $D$ , and collect their results  $R_j = D(Q_j)$  for each  $j \in [1, n]$ . (v) Compare the results of  $Q_0$  and  $Q_j$  to verify that  $R_0 \approx R_j$ , ensuring either equivalence or an acceptable level of variation, where  $\approx$  represents the predefined metamorphic relationship.

For example, as shown in Figure 1, the original MySQL query is deliberately mutated following a specific pattern to demonstrate an under-approximation. In this mutation, the “DISTINCT” keyword is strategically added to ensure uniqueness in the results, thereby removing any potential duplicates. This specific alteration helps highlight discrepancies between the original and mutated query results, which, if present, may indicate a logical bug due to a violation of the defined approximation relation.

**Motivation.** The dependency on grammar limits the adaptability of MOLTs. Specifically, no matter what generation model, SQL parser, or metamorphic mutator, they are all strongly dependent on the DBMS grammar, and adaptation to a new DBMS requires an amount of extra code, which is time-consuming and labor-intensive. Table 1 shows the lines of extra code of SQLancer, SQLRight, Pinolo and DQE to adapt for SQLite, PostgreSQL, MySQL, and MariaDB. This demonstrates that due to their dependency on grammar, all of these tools demand significant code expansion to accommodate different DBMSs. For example, SQLRight uses over 90,000 lines of code to build the generation model for PostgreSQL. In contrast, Pinolo simplifies its implementation by only supporting the common syntax shared between MySQL and MariaDB.

Table 1. **The lines of code of SQLancer, SQLRight, Pinolo and DQE to adapt for SQLite, PostgreSQL, MySQL, and MariaDB.** The numbers of lines of code are calculated from the official repositories of these tools. Due to the complexities in adapting the grammar, SQLRight does not support MariaDB, and DQE does not support PostgreSQL. Notably, although Pinolo supports both MySQL and MariaDB, it is implemented on a grammar common to both.

LoC	SQLancer	SQLRight	Pinolo	DQE
SQLite	10,020	17,120	-	7,606
PostgreSQL	8,012	93,671	-	-
MySQL	6,512	76,067	8,055	4,225
MariaDB	2,085	-	8,055	2,092

More importantly, there are 423 different DBMSs on the market [43], most of which use different grammar. Present-day leading MOLTs are designed primarily for well-known DBMSs such as MySQL, PostgreSQL and MariaDB. If we want to adapt them to these different DBMSs, it will take too long and require too many human resources to be economically feasible. Consequently, there exists a pressing need for a technique that allows for the straightforward extension of various MOLTs across multiple DBMS environments.



**Challenges.** Encouragingly, LLMs have shown stunning performance on various programming language tasks (e.g., sql generation [27, 51]), which points out a promising way to overcome challenges posed by syntactic differences across DBMSs. One potential solution is to leverage LLMs to translate SQL statement pairs, which are used in MOLTs, between different DBMSs; however, this approach presents significant challenges. Firstly, LLMs often lack a deep understanding of the dialect differences between DBMSs, which can lead to numerous syntactic and semantic errors [23], potentially resulting in functionally inequivalent queries. As shown in Figure 1, an LLM might translate a MySQL query “SELECT (COLLATION(`f`)) from t” to a simpler but incorrect “SELECT f from t” in PostgreSQL, omitting crucial dialect-specific functions like “COLLATION”, which are essential for character set comparison and sorting. Secondly, LLMs’ limited grasp of metamorphic mechanisms hampers their ability to maintain the necessary relationships between original and mutated queries, critical for identifying logical bugs. This deficiency can result in the loss of keywords such as “DISTINCT” so that the pair doesn’t preserve the intended metamorphic relationships, thus undermining the effectiveness of logical bug detection.

**Basic idea of QTRAN.** To address the first challenge, QTRAN leverages information from SQL documents to improve retrieval. We construct a detailed feature knowledge base from a variety of SQL documents in advance. These features are then mapped to their equivalents in the target DBMS using Retrieval-Augmented Generation (RAG), in which relevant snippets retrieved from a pre-established knowledge base further enhance the mapping answer. The enriched information, along with a customized prompt, is subsequently fed into the LLM to help the generation of semantically correct statements. To address the second challenge, QTRAN utilizes a mutation phase that deeply integrates metamorphic mechanisms into the LLM. We gather SQL statement pairs from existing fuzzers and fine-tune the LLM specifically for mutation tasks with these pairs. This phase helps LLMs to deeply understand the metamorphic mechanisms to mutate the original SQL queries.

Figure 2 illustrates a motivating example for how QTRAN extends MySQL queries to PostgreSQL. In the transfer phase, the MySQL original query “SELECT (COLLATION(`f`)) FROM t” is converted into a PostgreSQL-compatible form. The process begins by recognizing the “COLLATION” dialect in MySQL using an error recovery mechanism [30]. Next, we consult the pre-built feature knowledge base to map the “COLLATION” function to its PostgreSQL equivalent. This enriched knowledge is then fed into the LLM, which generates a PostgreSQL-compliant query while ensuring that the query’s semantics remain consistent. In the mutation phase, the customized LLM is used to generate mutated queries. This LLM is fine-tuned by incorporating a “FixDistinctL” mutation strategy. The fine-tuning process involves training the LLM on pairs of original and mutated queries, combined with explanations of metamorphic mechanisms, to ensure that the mutated queries adhere to the metamorphic testing principles crucial for identifying logical inconsistencies.

### 3 Approach

We propose an LLM-powered approach, QTRAN, to automatically extend existing MOLTs to various DBMSs. Inspired by the process of DBMS metamorphic testing, we decompose the analysis into two phases: the transfer and mutation phases. As shown in Figure 3, QTRAN starts with SQL statement pairs from existing MOLTs and extends these pairs to new DBMSs through the two phases.

- **Transfer Phase:** During the transfer phase, we start by identifying specific dialect features such as function names from the original queries using error recovery mechanisms. These identified features are then mapped to their equivalents in the target DBMS using a previously established feature knowledge base with the help of RAG. The enriched information and the original queries are then fed into the transfer LLM, which adapts these queries to the syntax of the target DBMS, ensuring accurate translation and semantic correctness.

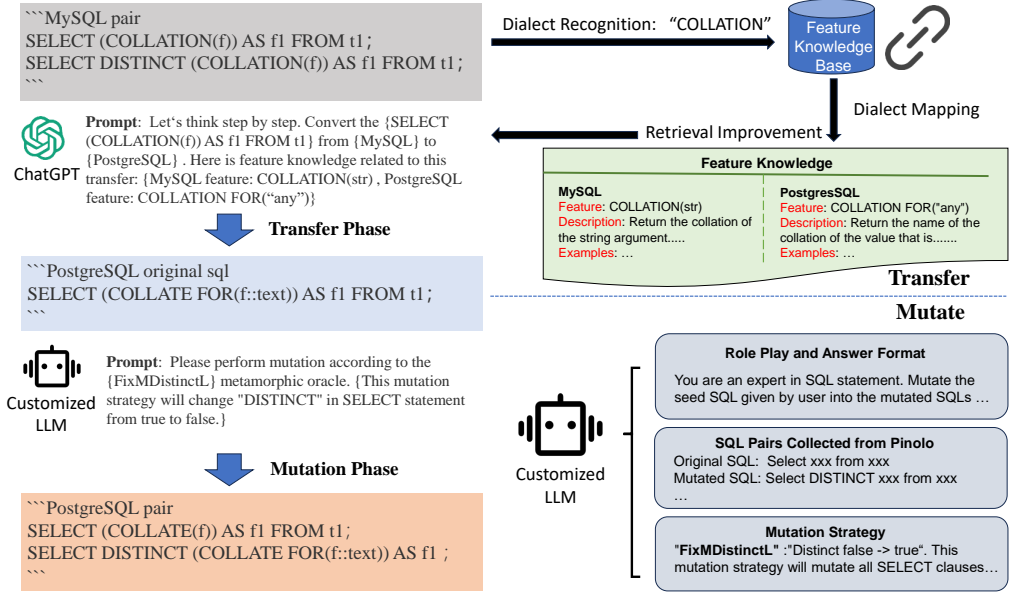


Fig. 2. **A motivating example illustrating how QTRAN transfers a SQL statement pair into the target DBMS.** The process of transferring a SQL statement pair involves two phases. In the transfer phase, original queries are translated into statements that comply with the syntax of the target DBMS. In the mutation phase, Customized LLM mutates these translated original queries.

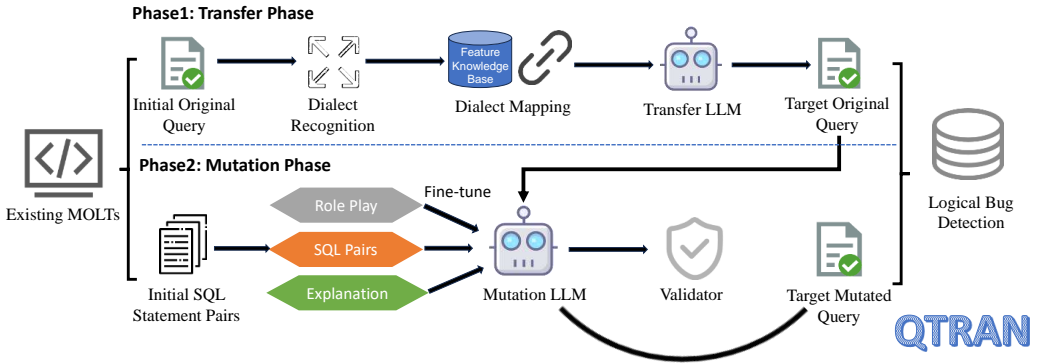


Fig. 3. **The workflow of QTRAN consists of two phases.** Transfer phase aims to transform the original queries in SQL statement pairs from popular DBMSs into statements that match the grammar of the target DBMSs. Mutation phase aims to further mutate these queries to ensure they maintain metamorphic relationships essential for detecting logical bugs.

- Mutation Phase:** In the mutation phase, we collect SQL statement pairs from existing MOLTs and enrich these pairs with role information and explanations of metamorphic mechanisms to fine-tune a mutation LLM. This tailored LLM is then used to mutate the original queries to fulfill metamorphic relationships, crucial for detecting logical bugs effectively. Additionally, we incorporate a validator to assess the effectiveness of the mutation LLM, ensuring that the mutated queries maintain their correctness and accurately reflect the intended logical transformations.

### 3.1 Transfer Phase

Transfer phase aims to transform original SQL queries from popular DBMSs to match the grammar of target DBMSs. We identify dialect-specific features from the original query, map them to the target DBMS using a feature knowledge base enhanced by RAG retrieval, and adapt the query to the target syntax via the transfer LLM to ensure accurate translation and semantic correctness.

**3.1.1 Feature Knowledge Base Construction.** To address the challenges of LLMs lacking deep understanding of DBMS dialects, leading to syntactic and semantic errors [23], we build a feature knowledge base from structured SQL documentation to enhance the LLM's comprehension of dialect-specific differences. The reason for extracting features from SQL documentation is that these documents are highly structured and contain detailed descriptions of the SQL syntax to each DBMS. Most databases provide comprehensive SQL documentation, making it an ideal source for constructing a robust and accurate feature knowledge base.

Next, we explain the process of constructing our feature knowledge base. We categorize the main SQL features, which exhibit distinct dialect characteristics, into three key types: data types, functions, and operators. It should be noted that these elements can be easily mapped across different DBMSs because they typically perform similar operations, such as string manipulation, mathematical calculations, or date handling, which are commonly required across all database systems. To ensure consistency and accuracy, we develop a dedicated crawler for each DBMS to extract the relevant features directly from their respective SQL documentation. Specifically, we first collect feature-HTML pairs following the official feature classification criteria. Then, we scrape information from the corresponding HTML pages for each feature and parse them into three key components: the feature rule, a clear description, and illustrative examples. We design personalized information parsing strategies for different databases, where the feature rules and illustrative examples are typically extracted from code-related elements, while the remaining text-related elements are considered as the clear description. Finally, we apply a data cleaning process including invalid data filtering, duplicates removal, and so on. This unified structure across the knowledge base allows the LLM to access enriched, organized dialect-specific knowledge, improving its ability to translate queries accurately between different DBMSs.

We also conducted a comprehensive survey of the features within the feature knowledge base. To the best of our knowledge, we are the first to perform an extensive investigation of dialect-specific features. As of the time of paper submission, we have constructed a feature knowledge base for eight popular DBMSs. Table 2 provides an overview of the feature knowledge base constructed for eight popular DBMSs, detailing the distribution of standard and dialect-specific features across three categories: data types, functions, and operators. Notably, dialect-specific variations in functions and data types account for a significant portion of the features in many DBMSs, reflecting substantial deviations from ANSI SQL standards. This observation underscores the importance of dialect mapping to ensure compatibility and accurate interpretation across different DBMSs.

**3.1.2 Dialect Recognition.** Dialect recognition is a crucial step in transferring SQL queries between different DBMSs. During this process, we leverage the error recovery mechanism provided by parsers ANTLR [34] to handle dialect-specific elements [30]. Specifically, QTRAN first breaks the content of the statement into a list of tokens, and then these tokens are sequentially fed to the standard parser. When the parser encounters an unrecognized or dialect-specific token, such as a function or data type unique to a particular DBMS, the error recovery mechanism allows the parser to skip over the problematic section and continue parsing the remaining tokens. These problematic tokens are precisely the objects for dialect mapping in the following steps. This approach enables



Table 2. **Statistics of feature knowledge base.** To determine whether a feature is dialect-specific, we employ a systematic approach. First, we construct a simple query for each feature using an LLM. Then, we parse this query with an ANSI standard parser. If the query fails to parse successfully, it indicates that the feature does not conform to the standard SQL grammar, identifying it as a dialect-specific feature.

DBMS	Data Types		Functions		Operators	
	Standard	Dialect	Standard	Dialect	Standard	Dialect
MySQL	25	8	229	249	73	6
MariaDB	18	8	193	208	39	6
TiDB	18	25	166	106	44	10
PostgreSQL	22	7	48	552	70	29
SQLite	28	8	49	78	22	5
MonetDB	19	8	51	181	38	8
DuckDB	27	18	88	368	11	10
ClickHouse	11	39	84	1,145	32	11

the system to identify parts of the query that require potential dialect-specific handling without prematurely terminating the parsing process.

**3.1.3 Dialect Mapping.** Dialect mapping is the process of translating features from the source DBMS to their equivalents in the target DBMS using the feature knowledge base. Since dialect-specific features, such as functions or data types, may vary significantly between DBMSs, we employ a RAG approach to accurately perform the mapping. The process starts with extracting key features, converting them into plain text, segmenting into chunks, and encoding them as vectors stored in separate vector databases for different DBMSs. For each dialect-specific feature in the source DBMS, similarity-based retrieval is used to find the most relevant features from the target DBMS's database. These retrieved features are then combined into an enhanced prompt to help the LLM generate the corresponding response. If a direct match is not found, we use vector similarity search to retrieve semantically related features. For instance, if a function like "COLLATION()" in MySQL needs to be translated, the system will retrieve a similar function by vector embedding similarity searching in the target DBMS, such as "COLLATE FOR()" in PostgreSQL, by leveraging the semantic knowledge (e.g. function name and description) stored in the feature knowledge base.

**3.1.4 Prompt Generation.** Once the potential dialect-specific features are identified, we utilize dialect mapping to enhance the retrieved knowledge, assisting the LLM in the transfer process. Our prompt is highly structured, designed to guide the LLM step-by-step in performing accurate SQL translations between DBMSs. As illustrated in Figure 4, the prompt includes three main components: (1) a description that defines the task and provides context for the LLM, (2) the original SQL statement in a code block for clarity, and (3) step-by-step commands using Chain-of-Thought (CoT) reasoning to ensure the translation maintains equivalent semantics and column names while adapting dialect-specific features.

## 3.2 Mutation Phase

In the mutation phase, we aim to mutate the original queries from the transfer phase according to the specific metamorphic mechanisms, which is essential for detecting logical bugs across different DBMSs. In this phase, we fine-tune the LLM with SQL statement pairs, which originate from existing MOLTs, along with enriched role-based information and explanations of metamorphic mechanisms. We also validate the customized LLM to ensure that it upholds the logical consistency and intended transformations necessary for effective logical bug detection. Next, we introduce two key components of the mutation phase: *Model Fine-tuning* and *Model Validator*.

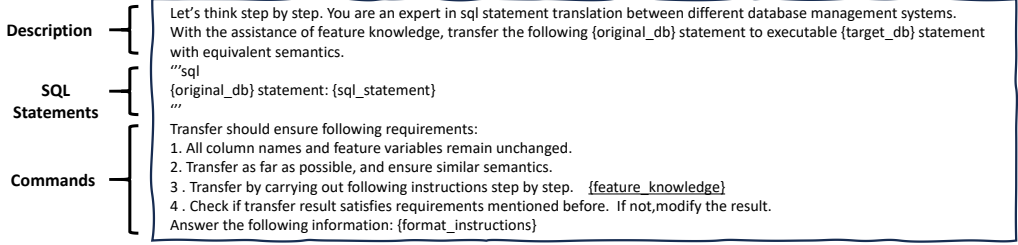


Fig. 4. **The transfer prompt template contains three main parts.** The parts in parentheses represent the variables in the prompt.

**3.2.1 Model Fine-tuning.** Fine-tuning is a crucial process that adapts pretrained LLMs to specialized tasks by further training them on domain-specific datasets, thereby improving their performance in targeted applications. This process customizes the general language capabilities of LLMs to excel in specific domains. Typically, we start with a pretrained LLM GPT-4o-mini, which has already learned a broad spectrum of language patterns and semantics from large text corpora. The model is then fine-tuned on a smaller, domain-specific dataset, transferring its general language knowledge to the mutation task [36].

The data for fine-tuning is sourced from existing MOLTs, which provide SQL statement pairs. Figure 2 illustrates the fine-tuning process using the “FixMDistinctL” mutation strategy. The fine-tuning process involves using a role-play format where the LLM is prompted as an expert in SQL mutation. The LLM is fed with SQL pairs, such as those collected from Pinolo, where the original SQL is mutated based on specific strategies like “FixMDistinctL”. By exposing the LLM to these structured pairs, it learns the intended logical transformations, enhancing its ability to detect logical bugs across various DBMSs. For each mutation strategy, we collected 20 SQL statement pairs, which were validated by the *Model Validator* to confirm their effectiveness, ensuring that the mutation LLM is exposed to a comprehensive variety of mutations during training [33]. QTRAN fine-tunes all the mutation strategies supported by the MOLTs before testing, ensuring that the model is well-prepared to handle any SQL statement pair during evaluation. It is worth noting that each MOLT may include multiple mutation strategies. For instance, Pinolo includes 25 mutation strategy, such as “FixMDistinctL”. To ensure the uniqueness of each mutation, we need to collect 500 (20\*25) SQL statement pairs for fine-tuning. This meticulous approach guarantees that every mutation is carefully crafted and tailored for specific logical transformations.

**3.2.2 Model Validator.** The purpose of the *Model Validator* is to ensure the effectiveness of the customized model after fine-tuning. We divide the collected SQL statement pairs into 80% for the training set and 20% for the validation set. During training, the model is assessed using multiple metrics, including training loss, training token accuracy, validation loss, and validation token accuracy. These metrics are computed both on a small batch of data at each step and on the full validation split at the end of each epoch. Once the training loss decreases to a stable point, and validation token accuracy reaches a sufficiently high threshold (above 95%), we can confidently use the customized model to mutate original queries, ensuring that the generated mutations maintain the intended logical transformations for detecting bugs.

## 4 Evaluation

This section presents a comprehensive evaluation of the effectiveness of QTRAN. The experiments conducted aim to answer the following research questions:

**Q1** Can QTRAN effectively transfer SQL statement pairs from existing MOLTs for use in metamorphic testing across various DBMS environments? What is the time cost associated with transferring SQL statement pairs using QTRAN?

**Q2** Can QTRAN find previously unknown bugs?

**Q3** How does dialect mapping help SQL translation?

**Q4** How does fine-tuning help mutation for metamorphic testing?

**Test DBMSs.** We focused on testing eight widely-used and large-scale open-source DBMSs: MySQL, MariaDB, TiDB, PostgreSQL, SQLite, MonetDB, DuckDB, and ClickHouse. Table 3 describes the detailed information of these databases.

Table 3. **The demographics of the DBMSs under test**

DBMS	Version	GitHub Stars	DB-Engine	First Release
MySQL	8.0.39	8.6K	2	1995
MariaDB	11.5.2	4.6K	15	2009
TiDB	8.3.0	33.1K	77	2017
PostgreSQL	16.3	16.1K	4	1989
SQLite	3.45.1	6.5K	10	2000
MonetDB	11.5.3	375	135	2004
DuckDB	1.1.2	23.6K	57	2018
ClickHouse	24.9.2.42	37.2K	37	2016

**Baselines.** In our evaluation, we selected four state-of-the-art MOLTs for extension: NoRec [37], TLP [38], Pinolo [20], and DQE [44]. Below, we detail each of these tools:

- (1) **NoRec** [37]: This technique involves transferring predicates from the WHERE clause to the SELECT clause. A logical bug is identified if the results differ after this transformation.
- (2) **TLP** [38]: This method decomposes a single query into three separate queries, each isolated by its predicates. It asserts a logical bug if the collective results of these queries diverge from the original query's outcome.
- (3) **Pinolo** [20]: Modifies query predicates to adjust constraints, expecting either a superset or subset of the original results, which serves to confirm the logic's soundness.
- (4) **DQE** [44]: This approach varies the types of SQL queries—such as SELECT, UPDATE, and DELETE—while keeping the predicates constant. It anticipates identical operations on rows and flags discrepancies as potential bugs.

These MOLTs require an understanding of the SQL syntax for different DBMSs. Table 4 describes the databases supported by these MOLTs. To avoid redundant testing, we employ QTRAN to extend these MOLTs and conduct tests on databases that are not natively supported by them.

Table 4. **Applicability of existing MOLTs for the selected DBMSs**

DBMS	NoREC	TLP	Pinolo	DQE
MySQL	✗	✓	✓	✓
MariaDB	✓	✗	✓	✓
TiDB	✗	✓	✓	✓
PostgreSQL	✓	✗	✗	✗
SQLite	✓	✓	✗	✓
MonetDB	✗	✗	✗	✗
DuckDB	✗	✓	✗	✗
ClickHouse	✗	✗	✗	✗

**Environment.** We conduct the experiments on one server with 104-cores Intel(R) Xeon(R) Gold 6230R CPU @2.10GHz and 500 GB memory. The server operates under the Ubuntu 20.04 OS, with the 5.4.0-135-generic version of the Linux kernel. If not otherwise specified, our experiments are conducted by invoking OpenAI API (version gpt-4o-mini-2024-07-18), with the proposed prompt in Section 3. To harness the deterministic nature of LLMs and ensure the generation of more predictable and stable outcomes, we set the temperature to 0.0.

#### 4.1 Q1. Effectiveness and Efficiency

In **Q1**, we aim to evaluate the effectiveness and efficiency of QTRAN in transferring SQL statement pairs from existing MOLTs across various DBMS environments. This is essential as it directly impacts the success of metamorphic testing.

##### 1) *Experimental Setup.*

We utilized QTRAN to extend four state-of-the-art MOLTs, denoted as QTRAN +NoREC, QTRAN +TLP, QTRAN +Pinolo, and QTRAN +DQE. These MOLTs were used to generate SQL statement pairs as source pairs for transfer. To generate the original SQL statement pairs, we selected the DBMS that each tool found the most bugs. Specifically, NoREC uses SQLite, DQE uses TiDB, while TLP and Pinolo use MySQL. Our focus was solely on databases not supported by the source MOLTs. In our experiments, a successfully transferred SQL statement pair is considered effective if it preserves syntactic and semantic correctness when executed on the target DBMSs, and fulfills predefined metamorphic relationships. We refer to such pairs as **valid pairs**. For SQL statements that fail during the transfer phase, we discard them immediately, foregoing the subsequent mutation phase. This strategy primarily impacts the efficiency of bug detection without causing false positives; we will discuss the impact of the transfer phase in detail in Section 4.3. For each evaluation, we ran QTRAN + MOLT (e.g., QTRAN +NoREC) for 5 hours on each target DBMS under test. For pairs that do not conform to the metamorphic relations, there is a potential indication of bugs. Subsequently, we verify whether they are true bugs through manual checks.

##### 2) *Results.*

Table 5 illustrates the overall results of QTRAN in transferring SQL statement pairs across unsupported DBMSs using the four extended MOLTs. Overall, the high percentage of valid pairs across all baselines, with most exceeding 99%, highlights the effectiveness of QTRAN in supporting metamorphic testing through efficient SQL transfer. In particular, QTRAN +DQE successfully transferred SQL statement pairs with a 100% valid pair rate on several DBMSs, including PostgreSQL, MonetDB and DuckDB. The consistent results across different baselines and DBMSs demonstrate that QTRAN can generate valid SQL statement pairs for metamorphic testing. For those SQL statement pairs that failed to satisfy the metamorphic relationships, we manually analyzed the potential bugs to exclude false positives. Figure 5 (a) shows the results of this analysis, illustrating that the overall false positive rate across baselines remains low. All baselines achieved a true positive rate of over 70%, with QTRAN +TLP reaching as high as 91%. Among the baselines, QTRAN +Pinolo has a slightly higher number of false positives, with 11 instances. This marginally higher rate may be due to the model's handling of more complex, nested SQL queries, which can occasionally lead to minor misinterpretations. It is important to note that false positives mainly result from hallucinations in the mutation LLM. To address this, we designed a validator (see Section 3.2.2) to ensure the model's effectiveness. When accuracy exceeds 95%, the false positive rate is significantly reduced, as this threshold indicates model convergence, ensuring generated mutations align with logical transformations and minimizing incorrect outputs.

Figure 5 (b) illustrates the average time distribution per SQL statement pair for the transfer and mutation phases across the four baselines. On average, QTRAN takes approximately 6.4

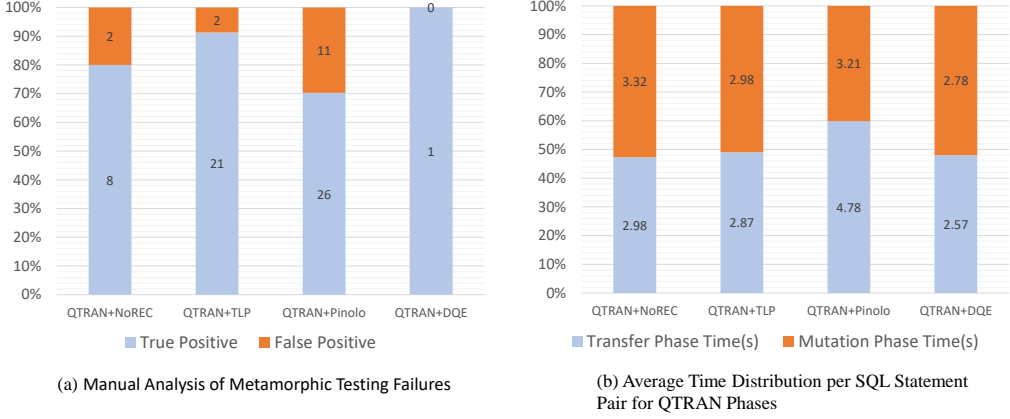


Fig. 5. **Transfer Efficiency and False Positive Analysis for QTRAN.** Sub-figure (a) shows the manual analysis of SQL statement pairs that failed to meet the metamorphic relationships, highlighting the number of false positives excluded in each baseline. Sub-figure (b) shows the distribution of time spent on transfer and mutation phases for each baseline.

seconds to generate a SQL statement pair. In particular, QTRAN +Pinolo exhibits a relatively higher time allocation for the transfer phase, averaging 4.78 seconds per SQL statement pair, suggesting that more time is required for complex query transformations. In contrast, QTRAN +NoREC, QTRAN +TLP and QTRAN +DQE demonstrate a more balanced time distribution between transfer and mutation phases, with transfer times of 2.98, 2.87 and 2.57 seconds, respectively. In summary, our approach enables the generation of valid SQL statement pairs across various DBMSs with a manageable time cost, supporting effective metamorphic testing.

Table 5. **Performance of QTRAN with extended baselines on unsupported DBMSs.** Column **Pairs** represents the number of generated SQL statement pairs, Column **Valid** indicates the number of valid pairs, and Column **Percent** is the percentage of valid pairs, calculated as  $\text{Percent} = (\text{Valid}/\text{Pairs}) \times 100\%$ .

DBMS	QTRAN+NoREC			QTRAN+TLP			QTRAN+Pinolo			QTRAN+DQE		
	Pairs	Valid	Percent	Pairs	Valid	Percent	Pairs	Valid	Percent	Pairs	Valid	Percent
MySQL	980	978	99.8%	-	-	-	-	-	-	-	-	-
MariaDB	-	-	-	1,000	992	99.2%	-	-	-	-	-	-
TiDB	993	985	99.2%	-	-	-	-	-	-	-	-	-
PostgreSQL	-	-	-	1,010	1,005	99.5%	685	678	99.0%	1,024	1,024	100.0%
SQLite	-	-	-	-	-	-	720	715	99.3%	-	-	-
MonetDB	897	897	100.0%	1,020	1,013	99.3%	730	727	99.6%	990	990	100.0%
DuckDB	1,049	1,049	100.0%	-	-	-	798	785	98.4%	1,043	1,043	100.0%
ClickHouse	997	997	100.0%	1,005	1,002	99.7%	776	767	98.8%	981	980	99.9%
<b>Total</b>	<b>4,916</b>	<b>4,906</b>	<b>99.8%</b>	<b>4,035</b>	<b>4,012</b>	<b>99.4%</b>	<b>3,709</b>	<b>3,672</b>	<b>99.0%</b>	<b>4,038</b>	<b>4,037</b>	<b>100.0%</b>

**Answer to Q1:** QTRAN demonstrates effective transfer of SQL statement pairs with high accuracy with an acceptable time cost across different DBMS environments, making it practical for extending MOLTs to support metamorphic testing in diverse database systems.

#### 4.2 Q2. Bug Detection.

In Q2, we aim to examine whether QTRAN can detect real-world bugs in DBMSs not supported by existing MOLTs, which serves as an indicator of the effectiveness of QTRAN.

##### 1) Experimental Setup.



We employ the same experimental setup as Q1. We ran QTRAN+MOLT for 5 hours on each target DBMS under test. We manually inspected potential bugs to exclude false positives and duplicate bugs, then submitted bug reports to the developers.

## 2) Results.

As shown in Table 6, we reported a total of 24 logical bugs across different DBMSs. Among these, 16 were confirmed by the developers, while 6 were identified as not bugs. Specifically, 9 bugs in MariaDB and 3 in TiDB were confirmed by the developers. Additionally, MySQL, PostgreSQL, DuckDB and ClickHouse each had 1 confirmed bug. Moreover, 2 bugs in ClickHouse are currently pending confirmation by developers. We continue to document true positive bugs and submit bug reports to developers, awaiting confirmation and feedback, a process that often requires a considerable amount of time. To keep track of the status of our reported bugs, we release the bug list in a public GitHub repository [45].

Table 6. **Summary of the logical bugs found by QTRAN.** The **Reported** column indicates the total number of bug reports submitted. The **Confirmed** column indicates the bug reports that developers have confirmed or fixed. The **Not a bug** column indicates the bug reports that developers have not confirmed or will not fix. The **Waiting** column indicates the bug reports that are pending confirmation.

DBMS	Reported	Confirmed	Not a bug	Waiting
MySQL	2	1	1	0
MariaDB	9	9	0	0
TiDB	4	3	1	0
PostgreSQL	1	1	0	0
SQLite	1	0	1	0
MonetDB	1	0	1	0
DuckDB	2	1	1	0
ClickHouse	4	1	1	2
<b>Total</b>	24	16	6	2

## 3) Case Study.


**True Positive Bug.** Figure 6 (b) shows a logical bug in DuckDB identified by QTRAN, triggered by the translation of the “DATE” function from SQLite to the “CURRENT\_DATE” function in DuckDB. The source statement pair is derived from the NoREC oracle, which detects bugs by moving the predicate from the “WHERE” clause into the “SUM” function, ensuring semantic consistency between the original and mutated statements. After the QTRAN transfer, however, the results of the original query and the mutated query in DuckDB diverged, indicating a logical bug.

Figure 6 (c) shows the results of directly transferring SQL statement pairs using GPT-4o-mini, which resulted in execution failures. The lack of dialect-specific knowledge within LLMs led to the erroneous translation of the SQLite “DATE” function to the “CURRENT\_DATE” function in DuckDB, causing the error “Scalar Function with name date does not exist!”. This highlights the superiority of QTRAN in translating SQL statement pairs, which better understands dialect nuances. In our bug list, 18 out of 24 bugs were overlooked due to the limitations of LLMs, highlighting QTRAN’s contribution to enhancing the reliability of DBMSs.

**False Positive Bug.** Figure 7 lists a representative bug that is classified as not a bug in TiDB. This bug, identified as MariaDB BUG #30299, was confirmed as a bug in MariaDB but not recognized as such by TiDB developers. The discrepancy arises due to different handling of SQL queries by MariaDB and TiDB, especially concerning implicit type conversion and the parsing of conditional expressions. Specifically, the condition  $\geq$  ANY in sql2 is more relaxed than the condition = ANY in sql1. In theory, the result of sql1 should be a subset of the result of sql2, which would indicate a bug

```
-- DDL
CREATE TABLE t0(c0);
CREATE TABLE t1(c0);
CREATE VIEW v0(c0) AS SELECT t0.c0 FROM t1 LEFT JOIN t0;
INSERT INTO t1(c0) VALUES ('example_value');
-- Source Original Query
SELECT COUNT(*) FROM v0 WHERE (DATE('now'), c0) != (CAST(NULL AS TEXT), '0'); --Result: {0}
-- Source Mutated Query
SELECT SUM(CASE WHEN (DATE('now'), c0) != (CAST(NULL AS TEXT), '0') THEN 1 ELSE 0 END)
FROM v0; --Result: {0}
```

(a) Source SQL Statement Pair in SQLite

```
-- DDL
CREATE TABLE t0 (c0 TEXT);
CREATE TABLE t1 (c0 TEXT);
CREATE VIEW v0 AS SELECT t0.c0 FROM t1 LEFT JOIN t0 ON t1.c0 = t0.c0;
INSERT INTO t1(c0) VALUES ('example_value');
-- Target Original Query
SELECT COUNT(*) FROM v0 WHERE (CURRENT_DATE, c0) != (CAST(NULL AS TEXT), '0'); --Result: {0}
-- Target Mutated Query
SELECT SUM(CASE WHEN (CURRENT_DATE, c0) != (CAST(NULL AS TEXT), '0') THEN 1 ELSE 0 END)
FROM v0; --Result: {1} 
```

(b) Target SQL Statement Pair in DuckDB Transferred by QTRAN

```
-- DDL
CREATE TABLE t0(c0 TEXT);
CREATE TABLE t1(c0 TEXT);
CREATE VIEW v0 AS SELECT t0.c0 FROM t1 LEFT JOIN t0 ON t1.c0 = t0.c0;
INSERT INTO t1(c0) VALUES ('example_value');
-- Target Original Query
SELECT COUNT(*) FROM v0 WHERE (DATE('now'), c0) != (CAST(NULL AS TEXT), '0');
--(duckdb.duckdb.CatalogException) Catalog Error: Scalar Function with name date does
not exist!
-- Target Mutated Query
SELECT SUM(CASE WHEN (DATE('now'), c0) != (CAST(NULL AS TEXT), '0') THEN 1 ELSE 0 END)
FROM v0;
--(duckdb.duckdb.CatalogException) Catalog Error: Scalar Function with name date does
not exist!
```

(c) Result of Direct SQL Statement Pair Transfer in DuckDB Using GPT-4o-mini

**Fig. 6. True Positive Bug: A logical bug in DuckDB when transferring source SQL statement pair in SQLite.** Sub-figure (a) demonstrates a SQL statement pair generated by NoREC in SQLite without errors. Sub-figure (b) demonstrates the target SQL statement pair in DuckDB transferred by QTRAN, resulting in an unexpected result due to a logical bug. Sub-figure (c) demonstrates the target SQL statement pair directly transferred using GPT-4o-mini, leading to translation failures due to limited understanding of the dialect difference between SQLite and DuckDB. The highlighted yellow background underscores the dialect differences between SQLite and DuckDB.

due to inconsistent execution results. However, TiDB developers have remarked that "The query is kind of ill-formed;" this inconsistency is due to different uses of implicit casting in the execution plans for  $=ANY$  and  $\geq ANY$ . The developers noted that "This is an uncommon syntax and is not recommended for use." Furthermore, in our bug list, we have identified 6 instances classified as "not a bug." These classifications are primarily due to the use of uncommon syntax by existing MOLTs, designed to trigger edge-case bugs, which some DBMS developers choose not to fix.

**Answer to Q2:** QTRAN detected a total of 24 logical bugs across various DBMSs, with 16 confirmed by developers, indicating its effectiveness in uncovering real-world bugs in DBMSs.

```

-- DDL
CREATE TABLE t (c1 FLOAT, c2 VARCHAR(20), key(c1));
INSERT INTO t VALUES (94.1106,'-0'),(1,'3 '), (0.0001,'-1');
-- sql1: Original Query
SELECT f1 FROM (SELECT c1 AS f1 FROM t) AS t1 WHERE ((-f1)=ANY (SELECT c2 FROM t)); --Result: 1
-- sql2: Mutated Query
SELECT f1 FROM (SELECT c1 AS f1 FROM t) AS t1 WHERE ((-f1)>=ANY (SELECT c2 FROM t)); --Result: Empty

```

Fig. 7. **Failure bug: Implicit casting and condition evaluation discrepancy in different DBMSs.** TiDB BUG #18560 found by QTRAN .

### 4.3 Q3. Contributions of Dialect Mapping

In Q3, we aim to investigate whether the key component of the transfer phase, dialect mapping, can help us in successfully translating SQL statements.

#### 1) Experimental Setup.

To understand the advantage of dialect mapping in the transfer phase, we implemented QTRAN<sup>-</sup> without dialect mapping. We also compared two state-of-the-art traditional rule-based SQL translation methods, namely SQLGLOT [32] and JOOQ [18], to highlight our contribution to SQL translation. SQLGlot is a lightweight SQL parser and transformer that supports the conversion of various SQL dialects, offering strong flexibility and scalability. JOOQ, on the other hand, is a Java library that facilitates database interaction by generating type-safe SQL queries, with support for multiple SQL dialect translations. We collected 100 original queries from each of four state-of-the-art MOLTs, denoted as QTRAN+NoREC, QTRAN+TLP, QTRAN+Pinolo, and QTRAN+DQE, and conducted transfer experiments on DBMSs not supported by these baselines. The initial DBMS selection for these baselines and the experimental configuration of Q1 are the same. We evaluated the transferred statements in terms of syntactic correctness and semantic correctness.

#### 2) Results.

As shown in Table 7, the semantic correctness ratios for QTRAN across various DBMSs are as follows: MySQL achieves 0.80, MariaDB scores 0.81, TiDB reaches 0.79, PostgreSQL improves to 0.74, SQLite records 0.75, MonetDB achieves 0.56, DuckDB scores 0.50, and ClickHouse reaches 0.53. Compared to QTRAN<sup>-</sup>, which represents the baseline without dialect mapping, these represent increases of +23%, +42%, +22%, +72%, +34%, +133%, +56%, and +141% respectively. By contrast, the semantic correctness ratios for SQLGLOT and JOOQ are significantly lower, with SQLGLOT achieving ratios ranging from 0.01 to 0.14 across DBMSs and JOOQ ranging from 0.05 to 0.32. These results underscore the limitations of rule-based methods in handling diverse SQL dialects compared to the LLM-based QTRAN. The results indicate that incorporating dialect mapping significantly improves the semantic correctness of SQL translations across different databases, underlining the effectiveness and utility of dialect mapping in enhancing translation quality in LLMs.

**Answer to Q3:** Dialect mapping significantly enhances the semantic correctness of SQL translations across different DBMSs by adjusting SQL queries to align with the specific syntax and functional behaviors of target databases.

### 4.4 Q4. Contributions of Fine-tuning

In Q4, we investigate the necessity of fine-tuning pre-trained models for mutation tasks. This section aims to clarify why QTRAN utilizes a fine-tuned mutation LLM to mutate transferred original queries, instead of directly transferring existing mutated queries from conventional MOLTs or relying solely on prompt engineering to perform mutations with LLMs.

Table 7. **The syntactic correctness ratios and semantic correctness ratios across various DBMSs under two baselines and two configurations.** QTRAN<sup>-</sup> represents the baseline without dialect mapping, and the improvement percentages in parentheses are relative to QTRAN<sup>-</sup>. Syntax correctness refers to whether the SQL statement can be parsed successfully by the database parser, while semantic correctness refers to whether the SQL statement can be executed successfully by the DBMS.

DBMS	Syntactic Correctness Ratios				Semantic Correctness Ratios			
	SQLGLOT	JOOQ	QTRAN <sup>-</sup>	QTRAN	SQLGLOT	JOOQ	QTRAN <sup>-</sup>	QTRAN
MySQL	0.35	0.70	0.87	0.95 (+9%)	0.13	0.25	0.65	0.80 (+23%)
MariaDB	–	0.67	0.84	0.90 (+7%)	–	0.24	0.57	0.81 (+42%)
TiDB	–	–	0.67	0.75 (+12%)	–	–	0.65	0.79 (+22%)
PostgreSQL	0.26	0.56	0.65	0.86 (+32%)	0.08	0.18	0.43	0.74 (+72%)
SQLite	0.37	0.76	0.75	0.82 (+9%)	0.14	0.32	0.56	0.75 (+34%)
MonetDB	–	–	0.58	0.76 (+31%)	–	–	0.24	0.56 (+133%)
DuckDB	0.25	0.55	0.68	0.83 (+22%)	0.04	0.16	0.32	0.50 (+56%)
ClickHouse	0.19	0.41	0.61	0.79 (+30%)	0.01	0.05	0.22	0.53 (+141%)

### 1) Experimental Setup.

To emphasize the importance of fine-tuning during the mutation phase, we designed two baseline configurations:

- **QTRAN-POM (Prompt-only Mutation of Transferred Queries).** In this configuration, we implemented QTRAN-POM without using a customized model. We directly apply the same base model, GPT-4o-mini with carefully crafted prompts to mutate transferred original queries. No fine-tuning or additional metamorphic relationship knowledge is incorporated into the model. This baseline examines whether prompt engineering alone is sufficient for generating valid metamorphic pairs without disrupting the predefined relationships between queries.
- **QTRAN-DTM (Direct Transfer of Existing Mutated Queries).** In this setup, we first mutate source original queries using existing MOLTs (e.g., NoREC, TLP, Pinolo, and DQE) and then directly transfer both original and mutated queries to the target DBMS in transfer phase. This baseline tests if existing mutated queries can be effectively adapted to the target DBMS without requiring an LLM-driven mutation process during transfer.

We selected 100 successfully transferred target original queries from each of the four baselines: QTRAN+NoREC, QTRAN+TLP, QTRAN+Pinolo, and QTRAN+DQE. We then performed mutations on these queries using QTRAN and two baselines, QTRAN-POM and QTRAN-DTM. We also employed the same evaluation method as in Q1. A valid SQL statement pair must ensure both syntactic and semantic correctness and fulfill predefined metamorphic relationships. For pairs that do not satisfy the metamorphic relationships, we conducted manual inspections.

### 2) Results.

Table 8 shows the percentage of valid pairs among QTRAN, QTRAN-POM, and QTRAN-DTM across four baselines. QTRAN-POM, which applies prompt-based mutation without fine-tuning, achieved valid pair rates below 50% across all baselines. QTRAN-DTM, which transfers existing mutated queries directly to the target DBMS, resulted in even lower valid pair rates, ranging from 0.08 to 0.34. The valid pair rates for QTRAN+NoREC, QTRAN+TLP, QTRAN+Pinolo, and QTRAN+DQE are 0.98, 1.00, 0.99, and 1.00, respectively. Compared to baseline methods, QTRAN achieves improvements of 120–376% over QTRAN-POM and 194–1138% over QTRAN-DTM, with the highest gains reaching 376% and 1138% against each baseline, respectively. These enhancements demonstrate the mutation LLM can help QTRAN effectively engage in metamorphic testing. By

fine-tuning with a small subset of SQL statement pairs from existing MOLTs, the LLM better grasps metamorphic mechanisms, thus significantly boosting its performance in mutating SQL statements across different DBMS environments.

We analyze the need for fine-tuning models in the mutation phase. The poor performance of QTRAN-DTM is due to several factors. First, transferring source and mutated queries separately to the target DBMS may lose the metamorphic relationship between them. While dialect mapping improves SQL translation success, it does not guarantee a one-to-one mapping for all dialect-specific components, and LLMs' inherent randomness causes variability in results. For example, the query `SELECT (COLLATION(f)) AS f1 FROM t1;` might be transferred as either `SELECT (COLLATE FOR(f::text)) AS f1 FROM t1;` or `"SELECT (COLLATE FOR(f)) AS f1 FROM t1;"`. Similarly, mutated queries can lead to inconsistencies, breaking the metamorphic relationship. Additionally, LLMs in complex queries often introduce hallucinations, generating incorrect components. For instance, QTRAN-DTM achieved only 0.08 valid query pairs on the QTRAN+Pinolo dataset, indicating frequent failures in maintaining metamorphic relationships. This results in missing SQL components or incomplete mutations, undermining testing accuracy and increasing false positives, making manual inspection inefficient. In contrast, fine-tuning the LLM for mutation tasks provides critical benefits by preserving metamorphic relationships between SQL query pairs, which are structured and consistent, making them well-suited for fine-tuning. Fine-tuned models demonstrate lower false positive rates through validation against training data, significantly reducing manual inspection overhead.

Table 8. **Comparison of the percentage of valid pairs among QTRAN, QTRAN-POM, and QTRAN-DTM across four baselines.** The numbers in parentheses represent the improvement percentages of QTRAN over QTRAN-POM and QTRAN-DTM, respectively.

Baselines	QTRAN+NoREC	QTRAN+TLP	QTRAN+Pinolo	QTRAN+DQE
QTRAN-POM	0.32	0.21	0.45	0.39
QTRAN-DTM	0.19	0.20	0.08	0.34
QTRAN	0.98 (+206%, +416%)	1.00 (+376%, +400%)	0.99 (+120%, +1138%)	1.00 (+156%, +194%)

**Answer to Q4:** Fine-tuning helps mutation for metamorphic testing by adapting the LLM to better understand the metamorphic mechanisms, ensuring accurate metamorphic relationships are maintained.

## 5 Discussion

**Usefulness and extensibility.** Our tool is highly user-friendly, allowing developers to seamlessly integrate MOLTs designed for one DBMS and extend them to other DBMSs using a straightforward and efficient method. The extensibility of our tool involves three primary efforts: ① Developers need to write a crawler to parse DBMS documentation and construct a feature knowledge base. This process typically requires around 200 lines of code, with the code size for the eight DBMSs we implemented ranging from 120 to 280 lines. ② Fine-tuning data must be collected from the SQL statement pairs provided by the source MOLTs. Using OpenAI's API, developers can fine-tune a customized LLM with minimal manual effort. ③ Verifying true positives from potential bugs requires around 5 minutes per bug by a person familiar with DBMS operations. In contrast, traditional methods require developers to have deep DBMS grammar knowledge and implement extensive code to create new generators and mutators for each DBMS. Our approach reduces this burden, providing a more efficient solution for extending MOLTs across various DBMSs.



**Limitations.** The limitations of QTRAN mainly lie in the following aspects: ① Although there is an improvement in the syntactic and semantic correctness of transferred queries compared to baselines (Table 7), the correctness ratios still have room for improvement. This is due to several factors. First, differences in DBMS feature support can lead to non-executable queries when the target DBMS lacks features present in the original query, such as unsupported functions. For example, MariaDB offers geospatial functions not found in other DBMSs, though this issue is rare according to our dialect mapping results. Second, LLM hallucinations can generate incorrect SQL statements. Notably, queries that fail during the transfer phase are discarded and do not affect the validity of the generated SQL pairs for metamorphic testing. ② The efficiency of QTRAN is closely tied to the response time of the underlying LLM. While the relatively slower response times of LLMs result in lower efficiency compared to traditional MOLTs, this trade-off underscores the advantage of automation, as traditional methods require substantial effort and costly adaptations by highly experienced experts to handle diverse scenarios.

**Threats to Validity.** In the implementation and evaluation of our system, we relied exclusively on the GPT-4o-mini model during the transfer and mutation phases, which may pose threats to external validity due to model-specific behaviors. Another potential threat is the evaluation methodology for valid mutations in metamorphic testing, where mutations may appear to fulfill metamorphic relationships but might not truly meet the criteria. However, our observations suggest that such instances are exceedingly rare and do not significantly impact our ability to test DBMSs effectively. Despite these threats, we have taken steps to mitigate them, and our findings provide valuable insights into using LLMs for extending MOLTs for logical bug detection in various DBMSs.

## 6 Related work

**Logical Bug Detection in DBMSs.** Researchers have proposed several methods for obtaining the test oracle to detect logical bugs, which can be categorized into three types: differential oracle, oracle-guided synthesis oracle and metamorphic oracle [20].

The first category is the differential oracles. Differential oracles [9, 13, 42] detect logical bugs by comparing the execution results of different DBMSs (or different versions of the same DBMS). However, existing studies [1, 13, 35, 37, 38, 42] highlight that differential testing is limited in its applicability, as not all DBMSs share the same SQL grammar or operation semantics, despite supporting the core SQL syntax, with each DBMS forming its own dialect [42], thus restricting the generality of differential testing. The purpose of QTRAN is to extend existing MOLTs to support various dialects, therefore differential oracles are not considered within the scope of this paper.

The second category is the oracle-guided synthesis approach [39]. PQS synthesizes a query that guarantees to return a specific row using its manually implemented interpreter. If the tested DBMS fails to fetch the row, PQS identifies a logical bug. Although such methods are tailored to the specific DBMS grammar, they only require constructing original queries to check the results. Therefore, the transfer phase of QTRAN can easily extend this approach.

The third category is the metamorphic oracles. To avoid the drawbacks of the differential oracle, many fuzzing methods [20, 28, 37, 38] use the metamorphic oracle to detect logical bugs. NoREC [37] creates a query by transferring predicates from the WHERE clause to the SELECT clause; the discovery of a logical bug occurs if results vary post-movement. Similarly, TLP [38] breaks down a single query into three, isolating each by its predicates, asserting a logical bug if the collective results diverge from the original query's outcome. SQLancer [3] integrates the above techniques and has been deployed to test various DBMSs. DQE [44] varies query types—such as SELECT, UPDATE, and DELETE—keeping predicates constant and anticipates identical row operations, flagging discrepancies as bugs. Pinolo [20] modifies query predicates to adjust constraints, expecting

either a superset or subset of the original results to confirm logic soundness. EET [25] transforms these expressions into semantically equivalent ones to construct queries that are semantically equivalent. QTRAN is an extension of these works. Due to their dependence on specific DBMS grammar, these tools are limited in adaptability. However, QTRAN successfully extends these works to new DBMSs by enhancing dialect knowledge and learning metamorphic mechanisms.

**LLM-based Fuzzing.** Following the success of LLMs in Natural Language Processing (NLP) tasks [10–12], the field of programming languages has seen significant advancements due to the integration of sophisticated AI-driven models for code generation and analysis. This development has naturally influenced fuzzing research: to help improve the fuzzing effectiveness, LLM has now become one of the key enablers to assist the core processes of fuzzing [14–17, 22, 31, 50]. Last year, the universal framework Fuzz4ALL [47] was introduced, which can target many different input languages and diverse features of these languages. These advancements have led to research into fuzzing techniques that employ LLMs, which are adept at producing syntactically informed, well-formed inputs for fuzzing [47, 49].

Different from these tools, QTRAN focuses on extending MOLTs for detecting logical bugs in DBMSs. While the aforementioned tools utilize LLMs to produce syntactically informed inputs for general fuzzing, they primarily detect easily observable error types, such as crashes and runtime errors. QTRAN enhances this approach by integrating an understanding of various DBMS metamorphic mechanisms and dialect knowledge, enabling the expansion of existing MOLTs to a wide range of DBMS grammar.

**Automatic grammar adaption.** Grammar adaptation is the process of adjusting the syntax and structure of code or statements to conform to the rules of a different language. Code translation has become a form of grammar adaptation, aiming to convert source code from one programming language to another. For example, tools like C2Rust [5] and CxGo [4] serve as transpilers converting C code to Rust and Go, respectively, while Sharpen [6] and Java2CSharp [7] facilitate the translation from Java to C#. On the topic of DBMS dialects, SQLess [30] achieves syntax-agnostic SQL query simplification by learning the syntax of different DBMSs through error recovery mechanisms. Unlike these tools, QTRAN is designed to transform SQL statement pairs containing SQL dialects, ensuring that the transformed pairs maintain metamorphic relationships. This approach allows QTRAN to support a wider range of DBMSs without the need for extensive grammar-specific configurations, making it a versatile tool for DBMS testing.

## 7 Conclusion

Due to the dependence of existing MOLTs on specific DBMS grammar, their extension capabilities are limited. In this paper, we proposed QTRAN, an innovative, LLM-powered approach that automatically extends existing MOLTs to various DBMSs. The evaluation results show that over 99% of the SQL statement pairs transferred by QTRAN satisfy the metamorphic relations required for testing, and have detected 24 logical bugs across several DBMSs, with 16 confirmed as unique and previously unknown. These results underline QTRAN's potential to significantly improve the reliability and testing robustness of diverse DBMSs.

## 8 Data Availability

The source code of QTRAN is available at <https://github.com/QTRANII/QTRAN>.

## Acknowledgement

We thank anonymous reviewers for their insightful comments. This work is supported by the Natural Science Foundation of China (62272400) and the fund from Ant Group. Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

## References

- [1] 2015. SQLsmith. <https://github.com/anse1/sqlsmith>. Accessed: 2024-10-24.
- [2] 2022. ANSI Standard. <https://www.ansi.org/>. Accessed: 2024-10-24.
- [3] 2023. SQLancer. <https://github.com/sqlancer/sqlancer>. Accessed: 2024-10-24.
- [4] 2024. C to Go Translator. <https://github.com/gotranspile/cxgo>. Accessed: 2024-10-13.
- [5] 2024. C2Rust Transpiler. <https://github.com/immunant/c2rust>. Accessed: 2024-10-13.
- [6] 2024. Java 2 CSharp Translator for Eclipse. <https://sourceforge.net/projects/j2cstranslator/>. Accessed: 2024-10-13.
- [7] 2024. Sharpen - Automated Java to C# Conversion. <https://github.com/mono/sharpen>. Accessed: 2024-10-13.
- [8] Atul Adya. 1999. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. (1999).
- [9] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases*. 1243–1251.
- [10] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
- [11] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.
- [12] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).
- [13] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [14] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.
- [15] Victor Dantas. 2023. Large Language Model Powered Test Case Generation for Software Applications. (2023).
- [16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.
- [17] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).
- [18] Lukas Eder. 2024. JOOQ: Fluent API for Typesafe SQL Query Construction and Translation. <https://www.jooq.org/>. Accessed: 2024-12-29.
- [19] Daniela Florescu, Alon Levy, and Alberto Mendelzon. 1998. Database techniques for the World-Wide Web: A survey. *ACM Sigmod Record* 27, 3 (1998), 59–74.
- [20] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 345–358.
- [21] William E Howden. 1978. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering* 4 (1978), 293–298.
- [22] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2023. Automated bug generation in the era of large language models. *arXiv preprint arXiv:2310.02407* (2023).
- [23] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, et al. 2024. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 492–496.
- [24] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. {DynSQL}: Stateful Fuzzing for Database Management Systems with Complex and Valid {SQL} Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4949–4965.
- [25] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 821–835.
- [26] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.
- [27] Zhenwen Li and Tao Xie. 2024. Using LLM to select the right SQL Query from candidates. *arXiv preprint arXiv:2401.02115* (2024).
- [28] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting logical bugs of {DBMS} with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. 4309–4326.

- [29] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. <https://www.usenix.org/conference/usenixsecurity22/presentation/liang>
- [30] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 743–754.
- [31] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. *arXiv preprint arXiv:2312.17677* (2023).
- [32] Toby Mao. 2024. SQLGlot: Python SQL Parser and Translator. <https://github.com/tobymao/sqlglot>. Accessed: 2024-12-29.
- [33] OpenAI. 2024. GPT-4 Turbo Fine-Tuning. <https://openai.com/index/gpt-4o-fine-tuning/>. Accessed: 2024-10-30.
- [34] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Chapter 9.
- [35] PingCap. 2023. go randgen. <https://github.com/pingcap/go-randgen>. [Online; accessed 2-October-2024].
- [36] Alec Radford. 2018. Improving language understanding by generative pre-training. (2018).
- [37] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.
- [38] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
- [39] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.
- [40] Manuel Rigger and Zhendong Su. 2022. Intramorphic testing: A new approach to the test oracle problem. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 128–136.
- [41] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.
- [42] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.
- [43] Solid IT. 2024. DB-Engines Ranking. <https://db-engines.com/en/ranking>. Accessed: September 3, 2024.
- [44] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing database systems via differential query execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2072–2084.
- [45] QTRAN Team. 2024. QTRANBugReports. <https://github.com/QTRANII/QTRAN-BUGLIST>.
- [46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [47] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [48] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing. *arXiv preprint arXiv:2408.15815* (2024).
- [49] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*. 435–450.
- [50] Cen Zhang, Mingqiang Bai, Yaowen Zheng, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2023. Understanding large language model based fuzz driver generation. *arXiv e-prints* (2023), arXiv–2307.
- [51] Qinggang Zhang, Junnan Dong, Hao Chen, Wentao Li, Feiran Huang, and Xiao Huang. 2024. Structure guided large language model for sql generation. *arXiv preprint arXiv:2402.13284* (2024).
- [52] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering* 34, 3 (2020), 1096–1116.

Received 2024-10-31; accepted 2025-03-31