Dialect-Agnostic SQL Parsing via LLM-Based Segmentation

JUNWEN AN, National University of Singapore, Singapore
KABILAN MAHATHEVAN, Virgina Tech, USA
MANUEL RIGGER, National University of Singapore, Singapore

SQL is a widely adopted language for querying data, which has led to the development of various SQL analysis and rewriting tools. However, due to the diversity of SQL dialects, such tools often fail when encountering unrecognized dialect-specific syntax. While Large Language Models (LLMs) have shown promise in understanding SQL queries, their inherent limitations in handling hierarchical structures and hallucination risks limit their direct applicability in parsing. To address these limitations, we propose SQLFlex, a novel query rewriting framework that integrates grammar-based parsing with LLM-based segmentation to parse diverse SQL dialects robustly. Our core idea is to decompose hierarchical parsing to sequential segmentation tasks, which better aligns with the strength of LLMs and improves output reliability through validation checks. Specifically, SQLFlex uses clause-level segmentation and expression-level segmentation as two strategies that decompose elements on different levels of a query. We extensively evaluated SQLFlex on both real-world use cases and in a standalone evaluation. In SQL linting, SQLFlex outperforms SQLFluff in ANSI mode by 63.68% in F1 score while matching its dialect-specific mode performance. In test-case reduction, SQLFlex outperforms SQLess by up to 10 times in simplification rate. In the standalone evaluation, it parses 91.55% to 100% of queries across eight distinct dialects, outperforming all baseline parsers. We believe SQLFlex can serve as a foundation for many query analysis and rewriting use cases.

CCS Concepts: • **Information systems** → **Query languages**; • **Software and its engineering** → **Parsers**.

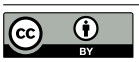
Additional Key Words and Phrases: SQL Dialect, Parser, Large Language Model

ACM Reference Format:
Junwen An, Kabilan Mahathevan, and Manuel Rigger. 2026. Dialect-Agnostic SQL Parsing via LLM-Based Segmentation. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 161 (June 2026), 28 pages. https://doi.org/10.1145/3802038

1 Introduction

Relational Database Management Systems (RDBMS) are among the most widely adopted data management platforms, with Structured Query Language (SQL) as the main interface for interacting with them. A core application of SQL is querying data. As a result, many important query-related tasks have emerged, which typically involve query analysis and rewriting. For example, SQL linting tools analyze queries for anti-patterns [15, 76], and may rewrite queries to fix such issues. Formally, we define *query rewriting* as transforming an input query Q into a new query Q' that satisfies a target objective O. *Query analysis* examines queries without modifying them. Since most rewriting begins with analysis, we use *rewriting* to refer to both for simplicity. Under this formulation, a wide range of applications beyond linting involve query rewriting, such as query reduction [47], DBMS testing [33, 50, 67], and SQL grading [8, 9].

Authors' Contact Information: Junwen An, National University of Singapore, Singapore, junwenan@u.nus.edu; Kabilan Mahathevan, Virgina Tech, Blacksburg, VA, USA, kabilan@vt.edu; Manuel Rigger, National University of Singapore, Singapore, rigger@nus.edu.sg.



ACM 2836-6573/2026/6-ART161
https://doi.org/10.1145/3802038

A major challenge in query rewriting is the diversity of SQL dialects, which leads to syntactic incompatibilities across systems [87, 91]. Most query rewriting tools follow a common workflow, but dialect-specific syntax frequently leads to failures. Specifically, these tools first parse an input SQL query with a grammar-based parser into an *Abstract Syntax Tree* (AST). The tools then analyze or modify the AST, and finally generate a query from the AST. The grammar-based parsers used in these tools often fail when encountering dialect-specific syntax, limiting their ability to support a wide range of DBMSs. For example, SQLFluff [76], a popular SQL linter, has more than 200 dialect-related open issues in its repository.

Grammar-based SQL parsers typically operate on a fixed set of grammar rules, and fail when encountering syntax unrecognized by the grammar. Some tools integrate grammar rules from multiple dialects into a single parser, allowing them to parse queries from multiple dialects into a unified AST. Notable examples include SQLFluff [76] and SQLGlot [80]. Although effective for the dialects they support, these tools require substantial manual effort to support new dialects. For example, although SQLGlot supports over 30 dialects, its entire parser codebase exceeds 20,000 lines, and adding support for a new dialect often requires over 1,000 additional lines. Additionally, existing dialects may introduce new features over time, requiring continued effort from such parsers' developers to support them. For instance, DuckDB recently integrated the `MATCH_RECOGNIZE` row pattern matching feature [37]. As a task-specific approach that tackles the SQL dialect problem, SQLess [47], a test case reducer, proposes an adaptive parser that attempts to generate new grammar rules when encountering dialect-specific features automatically. However, our experiments show that the generated rules fail to produce ASTs that can be interpreted by rewriting rules. Consequently, it remains a challenge to develop an automatic dialect-agnostic query parsing approach that generates an AST representation suitable for a wide range of query analysis and rewriting tasks.

Recent advances in Large Language Models (LLMs) have shown promise in SQL-related tasks [22, 40, 41]. Although LLMs demonstrate SQL understanding abilities, using them to directly generate an AST poses challenges. Their autoregressive nature makes them less effective at handling complex hierarchical structures, such as ASTs [30, 32, 57]. Additionally, end-to-end generation is prone to hallucinations [42], especially since no universal SQL grammar exists to validate outputs across dialects. We demonstrate these difficulties empirically in our evaluation.

In this paper, we take the first step toward addressing the limitations of grammar-based parsers and the drawbacks of using LLMs naively for dialect-agnostic query parsing. We aim to combine the accuracy of grammar-based parsing with the flexibility of an LLM-based segmenter to build an AST. To this end, we propose SQLFlex, a novel query rewriting framework that adopts this hybrid query parsing approach. While grammar-based parsers perform well at handling inputs that conform to predefined grammar, LLMs can interpret queries that include dialect-specific features. SQLFlex first attempts to parse a query using the grammar-based parser. When parsing fails due to dialect-specific features, SQLFlex invokes the segmenter to split the input into smaller parts such that they can be parsed or further segmented. This decomposes hierarchical parsing into sequential segmentation tasks that better align with the strengths of LLMs. Since clause-level grammar is typically flat, while expressions are recursive, SQLFlex employs clause-level segmentation and expression-level segmentation as two strategies suited to these different syntactic forms. Additionally, to improve reliability, SQLFlex validates the output by checking properties between the input and the output after each segmentation.

We conducted a large-scale evaluation of SQLFlex. Specifically, we showed SQLFlex's practicality in two real-world use cases, SQL linting and test-case reduction, each evaluated on a task-specific dataset. For linting, SQLFlex surpasses SQLFluff in ANSI mode by 63.68% in F1 score and performs on par with SQLFluff in dialect-specific mode (98.14% vs. 98.24%). For test-case reduction, SQLFlex outperforms SQLess by up to 10 times in simplification rate. Furthermore, we evaluated SQLFlex's

Listing 1. TSQL query, dialect-specific features highlighted

```
SELECT TOP 10 *
FROM Sales
WHERE (tot / 2) !< 8 AND year < 2025
OPTION (FAST 10)
```

Listing 2. Simplified ANTLR grammar of a query

```
selectStmt: selectClause fromClause? whereClause?;
selectClause: "SELECT" selectSpec? projections;
selectSpec: "DISTINCT" | "ALL";
fromClause: "FROM" tableReferences;
whereClause: "WHERE" expr;
expr: expr op expr | identifier | number;
op: "AND" | "<" | "/";
```

dialect-agnostic parsing effectiveness, using queries in eight different SQL dialects extracted from their respective DBMS test suites. SQLFlex successfully parsed 91.55% to 100% of queries across the eight dialects, outperforming both a PostgreSQL-specific parser and SQLGlot. On the most challenging dialect, SQLFlex achieves improvements of up to 179.46% over the PostgreSQL-specific parser and 138.04% over SQLGlot.

We believe that SQLFlex could be the foundation for many SQL-related use cases, where no manual effort is needed to support parsing of dialect-specific features. Like many other LLM-based applications (*e.g.,* Text-to-SQL [40, 41]), SQLFlex relies on a best-effort approach, so we defer exploring correctness-critical use cases such as SQL-level query optimization [7, 83, 90] as part of future work. To strengthen correctness guarantees, SQLFlex could potentially be integrated with query equivalence verification tools [31, 82].

To summarize, we make the following contributions:[1]

- We propose the novel idea of integrating the strengths of grammar-based parsers and LLMs for query parsing.
- We propose SQLFlex, a dialect-agnostic query rewriting framework which implements this idea with strategies for clause-level and expression-level segmentation.
- We extensively evaluated SQLFlex in two real-world use cases and in a standalone evaluation.

2 Background and Motivation

SQL standard and dialects. SQL is a standardized declarative language for data manipulation in RDBMSs [84]. We use *SQL standard* to refer specifically to SQL-92 [3], a minimal version *"supported by virtually all RBMSs"* [58]. The core operation in SQL is the query, typically the `SELECT` statement. Elements in a query can be grouped into *clauses* and *expressions*. A query is composed of multiple clauses, each of which specifies a particular aspect of the query. Clauses contain *clause elements*, which include both expressions and non-expression elements that contribute to the clause's behavior. Expressions are composable units made up of values, operators, functions, or subqueries, and they evaluate to a result [16]. Listing 1 shows an example query in the TSQL dialect [55], which contains the `SELECT`, `FROM`, `WHERE`, and `OPTION` clauses, where each clause includes the keyword and its associated elements. For example, within the `SELECT` clause, "`TOP 10`" is a non-expression clause element that limits the number of returned rows, and "`*`" is a wildcard expression selecting all columns. Similarly, the `WHERE` clause contains a predicate expression as a clause element that filters the result set.

[1]Our artifact is publicly available at https://github.com/wanteatfruit/SQLFlex and https://doi.org/10.5281/zenodo.18975512.

In practice, DBMSs implement their own variants of SQL, known as *SQL dialects* [58]. For example, SQL Server uses the TSQL dialect. Although the dialects share a common foundation (*e.g.,* SQL-92), they differ in grammar rules, reserved keywords, and DBMS-specific features [91]. For instance, unlike DuckDB, MySQL requires an alias for subqueries. In another case, "`OPTION`" has no special meaning in PostgreSQL, but in TSQL it is a reserved keyword and must be quoted when used as an identifier. A recent study found that test suites contain mostly dialect-specific features, where approximately 70% of queries in the PostgreSQL test suite are incompatible with other DBMSs, and about 60% for DuckDB [87]. In this paper, we focus on `SELECT` statements, and refer to syntax unsupported by the SQL standard as *dialect-specific features*.

Grammar-based parsers. Grammar-based parsers build an AST from an input query using a formal grammar. ANTLR [4] is a widely used parser generator that takes a grammar specification and generates a parser reflecting the grammar's structure. A grammar consists of *production rules* that define how symbols can be expanded. A *symbol* refers to any element in a production rule, which is either a *terminal* or a *non-terminal*. Listing 2 illustrates a simplified ANTLR grammar for a `SELECT` statement in the SQL standard. It denotes *terminals* in quotes and defines *non-terminals* using rule names. For instance, the rule `selectClause` includes the terminal `"SELECT"` and two non-terminals. An important observation from the SQL standard grammar is that clauses (*e.g.,* `selectClause`) tend to be non-recursive, while expressions are typically recursive.

Existing approaches. Most query rewriting tools rely on grammar-based parsers to construct an AST, but these parsers often fail on dialect-specific features, preventing further rewriting. We categorize grammar-based parsers as dialect-specific and multi-dialect parsers. For dialect-specific parsers, although they work well for the supported dialect, they lack generalizability across other dialects. Additionally, not all dialects have a ready-to-use parser in practice. This limitation also applies to parser generators like ANTLR, as grammars for many dialects are unavailable or incomplete [26]. In contrast, multi-dialect parsers aim to produce unified ASTs across dialects, offering better generalizability. Notable examples include the parser component in SQLGlot [80], SQLFluff [76], and Calcite [5]. While effective on multiple dialects, these tools demand substantial human effort and expertise to maintain the codebase and support additional dialects. For instance, the entire parser in SQLGlot contains over 20,000 lines of code, and adding support for an additional dialect often requires more than 1,000 additional lines. This high implementation cost makes such approaches difficult to scale, as evidenced by the many unresolved dialect-related feature requests in their open-source repositories [24, 25, 34].

3 Illustrative Example

We use the TSQL query in Listing 1 to illustrate the challenges of dialect-agnostic query parsing and how SQLFlex constructs its AST via hybrid segmentation. Dialect-specific features in the listing are colored, including the `OPTION` clause for query hints, the "`TOP 10`" select specifier, the not-less-than "`!<`" operator, and the use of "`year`" as an identifier despite being a reserved keyword in standard SQL. Grammar-based parsing alone fails here, as these features fall outside the grammar rules.

The core of our approach is *segmentation*, a divide-and-conquer strategy for handling dialect-specific features. When such features cause the grammar-based parser to fail, we prompt an LLM to decompose the query into multiple segments—aligning with the LLM's strengths in sequential processing [20, 32]—and iteratively process each segment using our hybrid approach. For example, when parsing the query in Listing 1, the grammar-based parser would fail, as "`TOP`" is an invalid terminal symbol in the lower-level `selectSpec` grammar rule. While it might be clear that "`TOP 10`" should be attributed to the `SELECT` clause rather than the `FROM` clause, as the `SELECT` clause would be incomplete otherwise, deciding so is difficult due to the English-like syntax of SQL [58, 73].

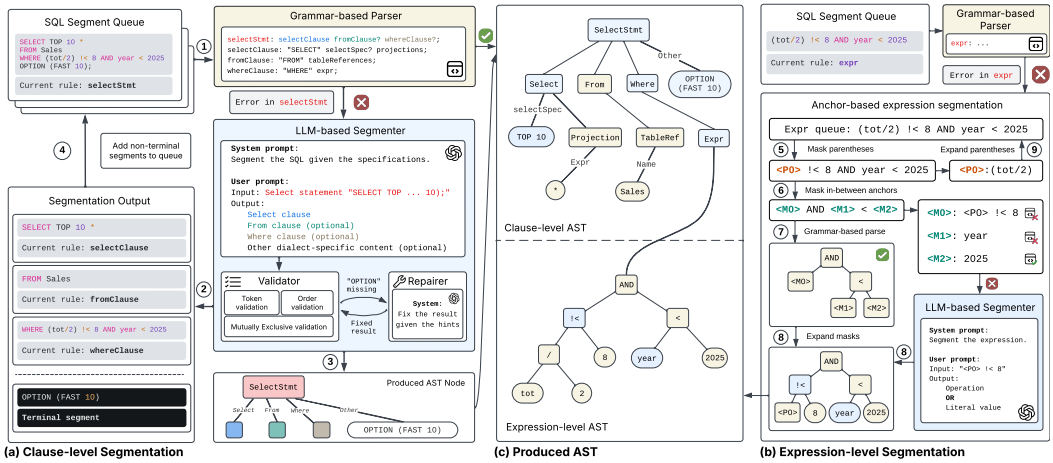


Fig. 1. Illustrative example of hybrid segmentation. Background colors of nodes denote their origins; text colors map grammar rules with segmentation prompts in (a).

Specifically, consider the `OPTION` keyword in Listing 1. In PostgreSQL, `OPTION` is accepted as an identifier, whereas in TSQL it marks the beginning of an `OPTION` clause. Without dialect knowledge, it is ambiguous whether `OPTION` should be interpreted as part of the `WHERE` clause or as a separate clause parallel to `WHERE`. Thus, rather than relying on heuristics, we use a state-of-the-art LLM that has been trained on examples of various SQL dialects to obtain more accurate parsing outputs.

Figure 1 illustrates our segmentation approach. When parsing the full query fails (①), we prompt the LLM to segment the query into three segments corresponding to the three non-terminals `selectClause`, `fromClause`, and `whereClause`, as well as dialect-specific elements that correspond to neither of the non-terminals (②). To enable this, we map non-terminals to prompts, in addition to a dedicated prompt for dialect-specific segments, each instructing the LLM to extract the corresponding segment from the input. These mappings are visualized using text colors in the figure. In the example, the returned segments are "`SELECT TOP 10 *`" for `selectClause`, "`FROM Sales`" for `fromClause`, "`WHERE...2025`" for `whereClause`, and "`OPTION (FAST 10)`", the latter of which is a dialect-specific segment that cannot be further parsed. After segmentation, we create the corresponding `SelectStmt` node (③) to build the AST top-down. The parsing process continues on the parseable segments (④). For example, in a subsequent iteration, the LLM segments the `SELECT` clause into "`TOP 10`" and "`*`", mapping them to `selectSpec` and `projections`, respectively. Since the `FROM` clause lacks dialect-specific features, grammar-based parsing is sufficient. We refer to this approach as *clause-level segmentation*, which we detail in Section 4.1. This approach is effective for non-recursive structures, which are present at the clause level.

At the expression level, clause-level segmentation is error-prone for deeply nested expressions. While most state-of-the-art LLMs perform well on simple cases, we observe that complex expressions are likely to be misinterpreted. For example, let `e` be the `WHERE` clause expression in Listing 1. When we wrap `e` inside a `CASE` expression (*e.g.,* `CASE WHEN e THEN e ELSE e END`), the LLM may misinterpret the "`AND`" operator in `e` as having the lowest precedence, overlooking the `CASE` expression. Additionally, applying clause-level segmentation to such expressions requires invoking the LLM at every level of the expression tree, causing a high computational overhead. To address this, our key observation is that certain operations, such as "`AND`" and "`<`", are recognized by the grammar, making their relative positions in the AST clear even in the presence of dialect-specific

features. Additionally, parenthesized subexpressions (*e.g.,* `(tot/2)`) indicate higher precedence and appear deeper in the AST, so we can defer their handling until the outer parts have been parsed. Motivated by these observations, we propose *expression-level segmentation*, a strategy designed to handle the expressions.

The core idea of *expression-level segmentation* is to treat known operators as *anchors* and replace fragments between them with *abstraction tokens*. This enables the grammar-based parser to parse the anchors and isolate the dialect-specific features. Figure 1b illustrates one iteration of expression-level segmentation, which begins when a query segment fails to be parsed using the expression-level `expr` rule. First, all parenthesized subexpressions are replaced with *parenthesis tokens* (*e.g.,* `(tot/2)` becomes `<P0>`) so that the current iteration focuses only on the non-parenthesized part (⑤). Next, in the remaining expression, recognizable operators (*i.e.,* anchors) "`AND`" and "`<`" are preserved, while the fragments between them are replaced with *mask tokens* (*e.g.,* `<P0> !< 8` becomes `<M0>`) (⑥). The abstracted expression "`<M0> AND <M1> < <M2>`" can now be parsed by the grammar, as dialect-specific features are isolated within mask tokens (⑦). After obtaining the AST of the abstracted expression, we expand each mask token (⑧), allowing the LLM to process only the fragments containing dialect-specific features rather than the full expression. Specifically, if grammar-based parsing fails for a token, we perform segmentation. For example, `<M0>` and `<M1>` cannot be parsed due to the dialect-specific `!<` operator and reserved keyword `year` used as an identifier. The segmenter identifies `<M0>` as an operation with "`!<`" as the operator and "`<P0>`" and "`8`" as operands. Once the dialect-specific operator is segmented, the operands can be parsed separately. For `<M1>`, the segmenter treats "`year`" as a literal, indicating that it is already a terminal node. `<M2>` is valid and parsed directly. Lastly, we expand the parenthesis tokens and add the enclosed expressions to the next iteration (⑨). For instance, "`tot/2`" in `<P0>` is processed in the next iteration. It contains no dialect-specific feature and is successfully parsed. We detail *expression-level segmentation* in Section 4.2.

Figure 1c shows the resulting AST after hybrid segmentation, where non-terminal nodes are visualized as rectangles and terminal nodes as ovals. Background colors indicate whether a node was produced by the grammar-based parser or the LLM-based segmenter. To improve reliability and reduce hallucinations, we apply validation and repair mechanisms to the segmentation outputs. The next section describes the overall approach in detail.

4 SQLFlex

We present SQLFlex, a dialect-agnostic query rewriting framework. We focus the rest of this section on detailing our key contribution, the hybrid segmentation process. As both AST manipulation and pretty-printing are based on standard techniques, we provide only a high-level overview of them in Section 4.4.

Algorithm 1 outlines the hybrid segmentation process, which builds the AST iteratively from top to bottom. Formally, given a query fragment Q, we define *segmentation* as a function $S : Q \rightarrow s_1, s_2, \ldots, s_n$, where each s_i is an output segment. We maintain a queue to manage query fragments whose sub-ASTs still need to be built; each entry is a tuple (Q, G), where G is the grammar rule used to parse Q. We initialize the queue with the query and `selectStmt` rule. In each iteration, we dequeue one item from the queue. Since grammar-based parsing is more efficient and reliable, we first attempt to parse Q using G (line 5). If parsing fails, the algorithm falls back on clause-level or expression-level segmentation depending on G (lines 7–10). Expression-level segmentation is used only when G is `expr`, indicating the parsing of an expression element; in all other cases, clause-level segmentation applies. Since clause-level segmentation produces intermediate segments that require further parsing, we add them to the queue. Both expression-level segmentation and grammar-based parsing construct the full sub-AST, so no additional fragments are enqueued.

Algorithm 1: Hybrid Segmentation

```
   Input: input_query                                              ▷ Full input query
   Result: root                                                    ▷ Parsed query AST
1  Function HybridSegmentation:
2      seg_queue ← [(input_query, selectStmt)], root ← ∅
3      while seg_queue not empty do
4          Q, G ← seg_queue.dequeue()
5          if parse(Q, G) succeeds then                            ▷ Grammar-based parsing
6              ast ← parse(Q, G)
7          else if G == expr then
8              ast ← ExpressionLevelSegmentation(Q)
9          else
10             ast ← ClauseLevelSegmentation(Q, G, seg_queue)
11         root ← attach(root, ast)
12     return root
13 Function ClauseLevelSegmentation(Q, G, seg_queue):
14     rule2ast, rule2prompt, segment2rule ← load_mapping()
15     N ← rule2ast[G]                                             ▷ AST node type for G
16     prompts ← rule2prompt[G]                                    ▷ Prompts associated with G
17     segments ← Segmentation(Q, prompts)
18     for s_i ∈ segments do
19         g_i ← segment2rule[s_i]                                 ▷ Maps each segment to one g_i
20         if has_nonterminal(g_i) then
21             seg_queue.enqueue((s_i, g_i))
22         else
23             N ← attach(N, s_i)
24     return N
```

4.1 Clause-level Segmentation

The core idea of clause-level segmentation is to map each non-dialect-specific segment s_i to a corresponding symbol g_i in the grammar rule G, so that s_i can be further parsed or segmented using g_i in subsequent iterations. Segments that lack a symbol mapping are treated as terminals. The clause-level segmentation function is also outlined in Algorithm 1, which takes as input the current query fragment Q, the associated grammar rule G, and the shared queue from the hybrid segmentation loop.

To support clause-level segmentation, we predefine three mappings (line 14). First, *rule2ast* maps each grammar rule G to its corresponding AST node N (line 15). For instance, `selectStmt` maps to a `SelectStmt` node. This information is needed to instantiate the correct AST node. Second, *rule2prompt* maps G to segmentation prompts, each associated with a symbol g_i (line 16). For example, `selectStmt` includes the `selectClause` symbol, mapping to a prompt *"Output should include a SELECT clause"*. We use few-shot prompting with examples of common SQL features to improve consistency and reduce bias toward a specific dialect. The prompts remain unchanged when applying SQLFlex across dialects. Lastly, *segment2rule* maps each output segment s_i back to its corresponding g_i, forming (s_i, g_i) tuples for the queue (line 19). For example, a segment corresponding to a `SELECT` clause will be mapped to `selectClause`. This helps the approach to resume parsing segments with the correct grammar rule.

After segmentation, each segment is processed based on the mapped g_i to determine whether it will be enqueued (lines 18–23). If g_i contains non-terminal symbols (*e.g.,* `selectClause` contains non-terminal `selectSpec`), the (s_i, g_i) tuple is added to the queue (lines 20–21). Dialect-specific features within these segments are handled in subsequent iterations to maintain sequential processing. In contrast, if g_i contains only terminal symbols, the segment is instead directly attached to the created AST node N as no further parsing is required (lines 22–23). This avoids repeated failures caused by dialect-specific features mapped to such rules. For example, in an iteration where Q is "`SELECT TOP 10 *`" and G is `selectClause`, the segment "`TOP 10`" maps to the terminal-only rule `selectSpec`, and is therefore attached to N directly without being enqueued. A special case occurs when a segment represents a dialect-specific feature lacking a corresponding g_i, such as the `OPTION` clause. These segments are encapsulated in dedicated `Other` terminal nodes and attached to N. We record their position in the query for correct pretty-printing. Section 6 shows that the core grammar suffices for various rewriting tasks, while content in `Other` nodes can be handled with text-based analysis. Our approach outperforms heuristic-based methods, as it preserves the structure of the AST for these nodes (*e.g.,* the `OPTION` clause is placed at the same level as the other clauses).

Clause-level segmentation returns the constructed AST node to the main hybrid segmentation loop (line 26). In subsequent iterations, the items added to the queue during clause-level segmentation are processed. If a dequeued item is $G = expr$ and parsing fails, the algorithm invokes expression-level segmentation.

4.2 Expression-level Segmentation

We propose an anchor-based strategy to isolate dialect-specific features into expression fragments that can be independently parsed. Specifically, the strategy leverages the precedence of parentheses and known operators (*e.g.,* `AND`) to construct partial ASTs, so that segmentation can be performed directly on their operands. We refer to it as *anchor-based expression segmentation*, outlined in Algorithm 2. The algorithm takes an expression as input and builds its AST in a top-down iterative manner. A queue manages the expression fragments to be processed, where each iteration processes one level of parentheses, starting with the input expression.

Given an expression E and a set of *anchors* (*i.e.,* known operators), the isolation process builds a partial AST T, where each leaf node contains a fragment e_i that may contain dialect-specific features. This divide-and-conquer approach allows each fragment to be segmented independently, preventing the LLM from misinterpreting nested subexpressions. This process consists of three steps, which we illustrate using the same expression in Figure 1b as E. First, we replace each parenthesized subexpression with a unique *parenthesis token* p_i, returning the transformed expression E' and mappings from each p_i to its original content (line 6). For example, `(tot/2)` in E is replaced with p_0, and we record the mapping $p_0 \mapsto$ `(tot/2)`. Next, we identify anchors in E' and replace surrounding content with *mask tokens* m_i, returning the transformed E'' and mappings from each m_i to its corresponding fragment e_i (line 7). For example, the anchors in E' are "`AND`" and "`<`", thus E' can be split into fragments e_i and anchors a_i as "$e_0 a_0 e_1 a_1 e_2$", where a_0 is "`AND`", e_0 is "`p0 !< 8`", etc. We replace each e_i with a mask token m_i and record their mappings. Lastly, we extend the base grammar to accept both token types, so that the masked expression E'' is parseable by the grammar-based parser. The produced T preserves the precedence of known operators (line 8). For dialect-specific operators whose precedence is unknown, our approach assumes they have the highest precedence among all unparenthesized operators, as low-precedence operators are typically reserved for common operators like "`AND`" and "`OR`" to compose multiple expressions. Determining the exact operator precedence is non-trivial both for the LLM and for users, as dialect-specific precedence rules are often inconsistent across DBMSs. In practice, we expect users to disambiguate

Algorithm 2: Anchor-based Expression Segmentation

```
   Input: input_expr                                              ▷ Full input expression
   Result: expr_root                                              ▷ Expression-level AST
1  Function ExpressionLevelSegmentation(input_expr):
2      expr_queue ← [input_expr], expr_root ← ∅
3      prompts, anchors ← load_prompts_and_anchors()              ▷ Predefined
4      while expr_queue not empty do
5          E ← expr_queue.dequeue()
6          E′, paren_map ← process_paren(E)
7          E″, mask_map ← process_anchor(E′, anchors)
8          expr_root ← parse_and_attach(E″, expr_root)            ▷ Produces T
9          for m_i, e_i ∈ mask_map do
10             if parse(e_i) succeeds then
11                 expr_root ← attach(parse(e_i), expr_root)
12             else
13                 expr_root ← attach(RecursiveSeg(e_i, prompts), expr_root)
14         for p_i, paren_content ∈ paren_map do
15             expr_queue.enqueue(remove_paren(paren_content))
16     return expr_root
```

the precedence of dialect-specific features that have a low precedence by adding parentheses, making operator precedence explicit, which is a best practice suggested by style guides [52].

We process each expression fragment e_i. For each masked fragment, the algorithm first attempts grammar-based parsing. Some fragments can be successfully parsed, as they may consist of recognized literal values or parenthesis tokens (including function calls where their argument lists are replaced by p_i). If parsing succeeds, the resulting subtree is attached directly to tree T (lines 10–11). If parsing fails, we recursively invoke the segmenter to handle the dialect-specific content and attach the resulting AST (lines 12–13).

At its core, the segmenter distinguishes between terminal and non-terminal fragments. Terminal fragments require no further parsing, while non-terminal fragments must recursively parse their child expressions to complete the structure. Thus, we prompt the LLM to output *either* a literal value segment or operation segments (*i.e.,* an operator and its operands). Literal segments are treated as terminal nodes (*e.g.,* `year`). For operation segments, the LLM identifies the operator with the lowest precedence (*i.e.,* the shallowest AST node) and its operands. Each operand is then recursively parsed or segmented. For example, given the fragment "`<P0> !< 8`", the LLM identifies `!<` as an operator, with operands `<P0>` and `8`, both of which can be parsed directly and attached to the tree. In more complex cases, such as "`1 !< 2 !< 3`", the initial segmentation identifies the first `!<` as a left-associative operator, with left operand "`1`" and right operand "`2 !< 3`". Since the right operand still contains a dialect-specific operator, segmentation is invoked again, continuing recursively until all terminal nodes are resolved.

After processing all fragments e_i, we handle the parenthesized expressions. For each such expression (*paren_content*), the outermost pair of parentheses is removed. The content is treated as a single expression and added to the queue for further expression-level segmentation (line 15). However, in two special cases, we proceed differently. First, if the content begins with "`SELECT`", we perform clause-level segmentation for the subquery. Second, if the content has unparenthesized commas—common in function arguments—it is treated as an expression list and split into separate

expressions, each enqueued individually. For example, given the function "`max(1, min(a, 1))`", we process its argument list. We remove the outer parentheses and split the arguments into `1` and `min(a, 1)`, enqueuing both. The comma in `min(a, 1)` remains parenthesized and will be handled in subsequent iterations. The algorithm continues until all parentheses are resolved (*i.e.,* no more expressions in the queue), and returns the expression-level AST. After combining with the clause-level AST, this yields the full AST for the input query.

4.3 Validation and Repair

For each segmentation output, regardless of the strategy, we apply three validation methods that check invariant properties between the input and the segmented output to detect potential errors. The LLM is prompted to repair its output until all validation checks pass. Note that while this approach improves robustness, it remains best-effort, as it might miss mistakes made by the LLM.

(1) *Token validation.* We compare the characters in each segment with those in the input to detect any missing or extraneous characters. For example, if "`OPTION`" is omitted during segmentation of the entire query, the validation fails.
(2) *Order validation.* We ensure that the character order within each segment matches the order in the original input. For example, if "`FAST 10`" is reordered as "`10 FAST`", the validation fails. We also check the semantic order among the segments. For example, in expression-level segmentation, if the left operand appears after the operator, the validation also fails.
(3) *Mutually exclusive output validation.* Expression-level segmentation defines mutually exclusive output types where the output is either a literal or an operation. If both output types are present, the validation fails.

Repair prompts are created based on the failed validation, the reason for the failure, and the full query to provide more context. After each repair attempt, the output is revalidated, repeating until success or reaching a predefined retry limit. If the limit is reached, we attach a special `Unsegmented` terminal node to the AST. This node preserves the original input and prevents further processing of that segment to maintain the overall structure of the AST. For example, if the `SELECT` clause fails to be segmented, an `Unsegmented` node containing the clause is attached to the `SelectStmt` node. The rest of the query (*e.g.,* `FROM` clause) can still be processed normally.

4.4 Rewriter and Pretty-Printer

While the *Rewriter* modifies the AST based on user-provided logic, the *Pretty-printer* regenerates the query from the AST. We consider neither of them novel and describe them only for completeness.

For the *Rewriter*, we provide two APIs to users: `find` and `transform`. The transform function is a higher-order API that applies a user-defined rule to modify AST nodes. It traverses the AST and applies the rule to each node. The `find` function complements this by searching for nodes of a specified type and returns nodes that satisfy an optional filter function. We designed the API to be minimal to support flexible and dialect-agnostic rewriting. Listing 3 shows an example of how users can detect and remove an unused table alias given an input query using SQLFlex. More complex AST manipulation can be implemented similarly by composing calls to `find` and `transform`, along with more extensive user-defined functions. Users can also incorporate dialect-specific knowledge to directly manipulate node attributes, including those representing dialect-specific features.

The *Pretty-printer* converts an AST back into its corresponding SQL query, ensuring that any modifications made by the *Rewriter* are reflected in the final output. It is a purely rule-based system that traverses the AST to regenerate the source text [12].

Listing 3. Detect and remove unused table alias

```
# detect anti-pattern using find API
def find_unused_table_alias(root):
  def is_table_alias(node): # filter function
    return isinstance(node.parent, Table)
  aliases = find(Alias, root, is_table_alias)
  identifiers = find(Identifier, root)
  if al in aliases and not in identifiers:
    return al

# user-defined rewrite rule for transform API
def rmv_table_alias(node, al):
  if isinstance(node, Table) and node.alias == al:
    node.alias = None

root = sqlflex_parse(query)
unused = find_unused_table_alias(root)
new = transform(root, rmv_table_alias, unused)
```

5 Implementation and Experimental Setup

We implemented SQLFlex in around 3,800 lines of Python. We adopted the SQL-92 [3] grammar as the base grammar. It has a minimal grammar that is supported by almost all SQL dialects [58], while allowing us to highlight the effectiveness of segmentation. We used ANTLR [4] to implement the grammar-based parser. We implemented the LLM-based segmenter with LangChain [38] and the OpenAI GPT-4.1 API [59], run at zero temperature for more deterministic outputs. We use a system prompt defining the general segmentation task (Listing 4) and allow up to three repair attempts to balance runtime and success rate. We used the LLM's structured output feature to ensure its outputs align with the prompts and can be subsequently processed. The output structure is defined using `pydantic`. For example, Listing 5 shows the structured output prompt for segmenting the `selectStmt` rule, where the `pydantic` model maps to the `SelectStmt` AST node definition. The `SelectStmt` class represents the input node type, and its attributes correspond to the expected output fields. Each field is annotated with a natural language description, which is automatically incorporated into the LLM prompt to guide structured generation. Note that the manual effort required to define these descriptions is incurred only once. We used LLMs to assist in generating and refining the descriptions, with the overall process taking less than ten minutes.

We ran all subsequent experiments on a server with a 64-core AMD EPYC 7763 processor (2.45GHz) and 512GB of RAM, running Ubuntu 22.04.

6 Use Cases

We present two real-world query analysis and rewriting use cases implemented using SQLFlex as the base parser, namely SQL linting and test-case reduction. SQL linting is a query analysis task, while test-case reduction involves query rewriting and requires executing the queries. The use cases highlight SQLFlex's practical utility, which grounds our evaluation in actual usage of SQLFlex before turning to a standalone evaluation.

6.1 SQL Linting

Many SQL queries exhibit poor coding practices that could cause performance and maintainability issues [71, 93], commonly referred to as SQL *anti-patterns* [35]. A prevalent example is the usage of the column wildcard expression (*e.g.,* `SELECT *`) to retrieve all columns from a table. While this

Listing 4. System prompt for LLM-based segmenter

```
You are an SQL expert in various SQL dialects. Your task is to segment an SQL query or a valid
part of a query into its components in the {dialect} SQL dialect while adhering strictly to the
following rules:
- Ensure all tokens from the original query are included in the segmented output. Do not add,
modify, or omit any tokens.
- Maintain the relative order of tokens as they appear in the input.

Important Notes:
- The input is always a valid SQL query or a valid fragment of a query in {dialect}.
- Use context information to understand the query.
- Reason step by step to ensure correctness and consistency: self-reflect on 3 to 5 results, and
return the most consistent result as the final output.
```

Listing 5. Structured output prompt fragment for `selectStmt`

```
class SelectStmt(BaseModel):
  """Represents a SQL SELECT statement."""
  select: str = Field(description="The SELECT clause, including the **SELECT**
      keyword and its associated projections.")
  from_: Optional[str] = Field(description="The FROM clause, including the **FROM
      ** keyword, specifying the source table(s).")
  where: Optional[str] = Field(description="The WHERE clause, including the **
      WHERE** keyword, defining filtering conditions.")
  other: Optional[str] = Field(description="Any additional dialect-specific
      contents.")
```

shorthand is convenient, it is discouraged in production, as it can lead to inefficiencies and cause schema changes to remain undetected [35]. Hence, a robust SQL linter is essential for detecting anti-patterns and improving code quality [15, 71, 76].

Baselines. We chose two notable existing linters, SQLCheck [15] and SQLFluff [76] as baselines. SQLCheck originally combined a non-validating parser with regular expression matching to detect anti-patterns [15]. However, its current open-source implementation relies only on regular expressions. SQLFluff uses a multi-dialect parser to parse input queries into ASTs, which are then analyzed for anti-patterns. While SQLCheck avoids parsing failures by using text-based analysis instead of ASTs, this approach leads to potentially lower accuracy. In contrast, SQLFluff has higher accuracy, but requires significant manual effort to support new dialects. Since SQLFluff allows users to configure the SQL dialect, we report results for its ANSI mode (*SF-ANSI*) and TSQL mode (*SF-TSQL*). Lastly, we implemented a purely LLM-based baseline using GPT-4.1 to investigate whether an end-to-end LLM-based approach is sufficient for this task. We provided the LLM natural language rule descriptions and prompted it to output anti-patterns.

Rule selection. We systematically selected a set of fourteen linter rules for the evaluation, presented in Table 1. We first selected all three rules that are supported by both SQLFluff and SQLCheck. For a more comprehensive evaluation, we incorporated eleven additional rules from SQLFluff. We chose these rules from SQLFluff's *core* rule set, which are defined by the developers as *"Stable, applies to most dialects, could detect a syntax issue, and is not too opinionated toward one style"* [76]. Within the core rule set, we excluded those addressing layout issues (*e.g.,* incorrect indentation),

Table 1. List of Selected Linter Rules

Source	ID	Description
SQLFluff	AM01	Ambiguous use of `DISTINCT` with `GROUP BY`.
	AM06	Inconsistent column references in `GROUP/ORDER BY` clauses.
	AL02	Implicit aliasing of columns.
	AL03	Column expression without alias.
	AL04	Table aliases should be unique within each clause.
	AL05	Tables should not be aliased if that alias is not used.
	AL08	Column aliases should be unique within each clause.
	AL09	Column aliases should not alias to themselves.
	CV04	Use consistent syntax to express "count number of rows".
	CV05	Comparisons with `NULL` should use `IS` or `IS NOT`.
	RF01	Referencing objects not present in the `FROM` clause.
Both tools	AM02	`UNION [DISTINCT|ALL]` is preferred over `UNION`.
	AM04	Query produces an unknown number of result columns.
	AM08	Implicit cross join detected.

symbol inconsistencies (*e.g.,* extra commas), and capitalization inconsistencies (*e.g.,* keyword capitalization), as SQLFlex does not preserve this information during parsing. For brevity, we will refer to each rule by its ID in Table 1 in subsequent paragraphs.

Dataset. We selected the SESD dataset [29], which consists of human-authored SQL queries with diverse TSQL-specific features collected from *Stack Exchange*, and we expected some of them to include anti-patterns targeted by SQL linters. We removed duplicate queries and excluded queries that failed to parse using a TSQL parser. Finally, we obtained 1,916 TSQL queries, including 111 queries with window functions and 1,030 queries with joins.

Since SQLFluff already supports the TSQL dialect, we constructed reliable ground-truth annotations (*i.e.,* the list of potential anti-patterns for each query) by initially running both SQLFlex and SQLFluff on the entire dataset. Cases where both tools detected the same anti-patterns were considered the ground-truth. For discrepancies between the two tools, we manually reviewed and confirmed each instance to ensure accurate labeling. To further validate the constructed ground truth, we randomly sampled 50 labeled queries and manually annotated them. We observed no discrepancies between these annotations and the constructed labels. Based on this sample, a Clopper–Pearson confidence interval shows that, with 95% confidence, the true error rate is below approximately 6%.

Metrics. We use *Precision*, *Recall*, and *F1 score* as evaluation metrics [68]. The metrics are computed at the anti-pattern level. For example, if the ground truth for a query is {AM01, AL02, RF01}, but SQLFlex reports {AM01, AL03}, we count one true positive (AM01), one false positive (AL03), and one false negative (RF01). Here, true positives refer to correctly detected anti-patterns, false positives to incorrectly reported ones, and false negatives to missed anti-patterns. For all three metrics, higher values indicate better performance. Since SQLCheck supports only three of the selected rules, we compute its metrics based on those three rules.

Results. Figure 2 shows the overall results for all selected linter rules. SQLFlex significantly outperforms the purely LLM-based approach, SQLCheck, and *SF-ANSI* across all three metrics. The purely LLM-based linter suffers from high false positive rates, likely due to the ambiguity of natural language rule descriptions and the lack of effective validation mechanisms, which highlights the importance of AST-based analysis. Similarly, SQLCheck, which relies on text-based analysis without accounting for SQL's structural semantics (*e.g.,* column names), also performs poorly. *SF-ANSI*

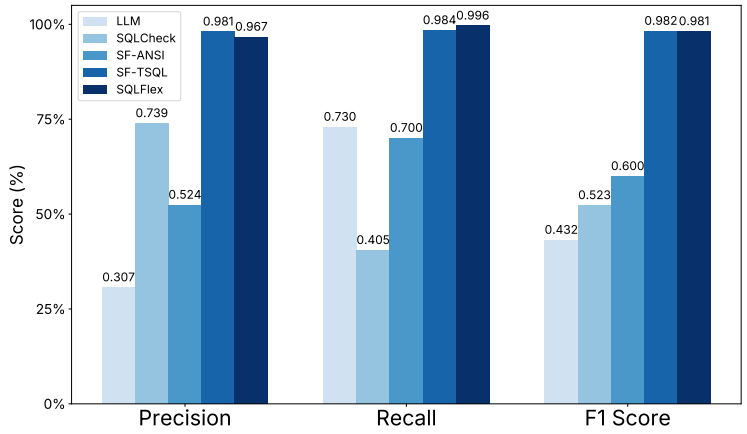


Fig. 2. Overall performance comparison of linters

frequently failed to parse queries in the SESD dataset due to TSQL-specific syntax, such as using square brackets to enclose identifiers with special characters or reserved keywords (*e.g.,* `SELECT t AS [SELECT]`), leading to low linting accuracy. In contrast, SQLFlex was able to handle such syntax correctly via the hybrid segmentation approach.

When compared with *SF-TSQL*, SQLFlex achieves slightly lower precision (96.69% vs. 98.06%), higher recall (99.63% vs. 98.42%), and comparable F1 score (98.14% vs. 98.24%). The errors made by SQLFlex are mainly due to imprecise segmentation by the LLM when faced with ambiguous syntax in queries. For example, in "`SELECT Count FROM t ...`", the LLM misinterpreted `Count` as an expression (resembling function `COUNT()`) rather than a column name, causing the linter to incorrectly report the anti-pattern "AL03". *SF-TSQL* also had occasional parsing failures that led to incorrect results. For example, it failed to parse the query "`SELECT Class AS Class ...`", causing it to miss the anti-pattern "AL09". Overall, the linter implemented using SQLFlex matched the performance of SQLFluff, with significantly less manual effort.

6.2 Test-Case Reduction

Test-case reduction is a query rewriting application. Automated DBMS testing tools, such as SQLancer [75] and SQLsmith [2], often generate long and complex queries to expose bugs. However, these queries are often too complicated for developers to identify the underlying issues. Consider the NoREC oracle [66], which identifies logic bugs by comparing results between two semantically equivalent queries that differ only in whether query optimization is applied. The resulting bug reports typically contain complex expressions [46, 66, 88] that make it difficult for developers to fix potential bugs. Therefore, reducing test cases while preserving bug-triggering behavior is essential for efficient bug-fixing [60, 72, 77].

Baselines. SQLess [47] is a state-of-the-art test case reducer designed specifically for dialect-agnostic SQL reduction. It uses AST-based strategies, such as removing clauses and simplifying expressions, to iteratively reduce queries. After each reduction, SQLess executes the reduced query with the test oracle to verify whether the bug can still be triggered. When encountering a query containing dialect-specific features, SQLess uses ANTLR's syntax-based error recovery algorithm to generate new grammar rules. With these added grammar rules, SQLess can parse previously unsupported queries. We compare a test-case reduction tool implemented in SQLFlex with SQLess as the baseline.

Table 2. Simplification ratios by DBMS

DBMS	MaxSimRatio		AvgSimRatio	
	SQLess	SQLFlex	SQLess	SQLFlex
MySQL	**87.92%**	**87.92%**	6.27%	**7.53%**
SQLite	86.83%	**94.75%**	1.83%	**18.91%**

Dataset. We constructed a new dataset of bug-triggering queries in SQLite and MySQL to evaluate SQLFlex. A possible alternative would have been SQLess's dataset. However, it is not publicly available and is imbalanced, containing over 32,000 MySQL-compatible cases, but only 143 SQLite cases. Thus, we used SQLancer, a widely used DBMS testing tool, to generate bug-triggering queries in prior versions of SQLite (v3.28.0) and MySQL (v8.2.0). We increased the test case complexity by setting SQLancer's expression-depth parameter to 5, then collected the first 1,500 bug-inducing test cases for each dialect, retaining only those found by the NoREC oracle. The final dataset includes 1,166 SQLite and 1,491 MySQL test cases, among which 583 contain subqueries, and 767 have joins. Notably, SQLess already supports MySQL as it uses the MySQL grammar, but needs its adaptive parser to handle SQLite-specific features.

Implementation. We adopted the test-case reduction algorithm described in SQLess [47]. Specifically, we implemented expression reduction strategies (*e.g.,* removing the left operand of a logical operator) and clause reduction strategies (*e.g.,* removing the `ORDER BY` clause) on the ASTs generated by SQLFlex. We implemented only the strategies relevant to our dataset and sufficient for effective simplification. Since SQLess applies strategies enumeratively, this selective implementation does not bias the results. For a fair comparison, we disabled additional strategies in SQLess.

Metrics. Following SQLess, we used *Average Simplification Ratio* (AvgSimRatio) and *Maximum Simplification Ratio* (MaxSimRatio) as evaluation metrics [47]. These metrics measure the percentage reduction in the number of tokens between the original and reduced queries, with higher values indicating a more effective reduction.

Results. Table 2 compares the simplification ratios achieved by SQLess and SQLFlex. Overall, SQLFlex performs better than SQLess. For the MySQL test cases, both tools achieved comparable performance, showing that SQLFlex, despite using a minimal grammar—with 61.82% of reduced queries requiring segmentation—can match the effectiveness of a full-grammar-based tool. For the SQLite test cases, SQLFlex significantly outperforms SQLess. This is primarily because SQLess struggled with SQLite-specific features, while SQLFlex effectively handled them. The adaptive parser in SQLess appends all unrecognized tokens as terminal symbols without capturing their semantic meaning. Moreover, for complex queries, the adaptive parser often generates incorrect grammar rules, resulting in ineffective reductions, as the reduction rules fail to interpret the tree nodes. In contrast, although 92.78% of reduced queries were not supported by SQLFlex's grammar, SQLFlex still generates ASTs that preserve the semantic structure of the SQL query. Its anchor-based expression segmentation approach further enables robust handling of complex expressions, leading to more effective reductions.

7 Standalone Evaluation

In this section, we further investigate SQLFlex itself, that is, whether SQLFlex can effectively parse queries. Specifically, we aimed to answer the following questions:

- **Q1.** How effectively does SQLFlex handle dialect-agnostic parsing of SQL queries (see Section 7.1)?

Table 3. List of datasets and query features used in the standalone evaluation (WF: Window Function, CTE: Common Table Expression, SubQ: Subquery)

Dataset	Queries	WFs	CTEs	SubQs	Joins
DuckDB Test Suite	2,092	185	22	496	284
PostgreSQL Test Suite	1,207	139	0	162	128
MySQL Test Suite	1,039	3	0	67	77
ClickHouse Test Suite	2,485	69	1	573	513
H2 Test Suite	358	21	0	67	50
CockroachDB Test Suite	1,369	100	2	287	202
Cassandra Test Suite	150	0	0	0	0
SurrealDB Test Suite	351	0	0	5	0
Total	**9,051**	**517**	**25**	**1,657**	**1,254**

- **Q2.** How do SQLFlex's key components contribute to its overall effectiveness (see Section 7.2)?
- **Q3.** How efficiently does SQLFlex handle SQL queries with dialect-specific features (see Section 7.3)?
- **Q4.** Can SQLFlex be extended to parse Data Definition Language (DDL) statements, such as `CREATE TABLE` (see Section 7.4)?

Dataset. To investigate whether SQLFlex can handle diverse and niche dialect-specific features, we built a large-scale benchmark dataset extending the dialect types covered in Section 6. To the best of our knowledge, there is no publicly available benchmark suitable for the evaluation of dialect-agnostic SQL parsing. While there are existing query benchmarks such as TPC-H and SQLStorm [69], these benchmarks are designed for DBMS performance evaluation and primarily contain standard SQL features. Our benchmark consists of queries of eight different dialects, which we collected from the corresponding DBMS's test suite. These test suites typically include statements that reflect the unique features of their respective systems [87], making them more suitable for the evaluation of SQLFlex's dialect-agnostic parsing ability.

We selected the SQL dialects and the DBMSs that implement them based on several factors: (1) availability of open-source test suites; (2) diversity in usage and functionality, including a combination of well-established systems (*e.g.,* PostgreSQL and MySQL) and emerging or specialized ones (*e.g.,* DuckDB and SurrealDB) [14]; (3) distinct dialect-specific features, avoiding largely compatible systems like MySQL and MariaDB that share a common codebase. Table 3 summarizes the datasets, including the number of queries and the occurrence of complex query features (*i.e.,* window functions, CTEs, subqueries, and joins). The datasets cover a broad range of SQL features, evaluating SQLFlex comprehensively.

We built the benchmark dataset by extracting `SELECT` statements from test suites. To ensure syntactic validity, we excluded queries that failed to parse using dialect-specific parsers. Since many of the extracted queries had similar structures or features, we applied a keyword-based deduplication heuristic to retain a diverse subset. SQL parsing depends primarily on keywords [73], making keyword diversity a useful proxy for syntactic variety. We treated queries with identical keyword sets as duplicates, retaining only the longest one, assuming it to be the most complex. For H2, since we were unable to obtain a ready-to-use H2-specific parser, we used JSQLParser, which supports H2 syntax. For SurrealDB, as we were unable to obtain its keyword list, we kept all queries in its dataset.

7.1 Q1: Effectivness and Reliability

In this section, we show SQLFlex's effectiveness and reliability in parsing queries in diverse SQL dialects.

Baselines. We compared SQLFlex against representative baselines. To the best of our knowledge, no existing academic work has proposed a general-purpose, dialect-agnostic SQL parser. The most closely related work is SQLess [47]. As SQLess is applicable only to test-case reduction, we evaluate it on this use case in Section 6.2. Thus, we compare SQLFlex against two open-source tools, which are representatives of dialect-specific parsers and multi-dialect parsers, and a purely LLM-based baseline parser:

- **PostgreSQL Parser:** We chose the PostgreSQL parser as the dialect-specific parser baseline. We used `pglast` (v6.10) [62], a Python package of the official PostgreSQL parser. While dialect-specific parsers exist for other dialects, we chose to focus on PostgreSQL for two reasons. First, it is a widely used dialect with strong conformance to SQL standards. Second, we expect similar results for other dialect-specific parsers, as they share the same limitation. Specifically, we expect them to perform perfectly on their target dialect, but fail to parse specific features of other dialects.
- **SQLGlot v25.22.0:** SQLGlot [80] is a state-of-the-art multi-dialect parser supporting over 30 dialects. We evaluated two configurations of SQLGlot, which are its standard mode (**SQLGlot**$_s$), which parses queries using a superset of all supported dialects, and its dialect-specific mode, where the dialect is supported (**SQLGlot**$_d$). SQLGlot is expected to perform well on its explicitly supported dialects due to its extensive manual implementation effort, and better than `pglast` on unsupported dialects in its standard mode. In other words, we set SQLGlot's configuration so that it is expected to perform best for each of the dialects.
- **LLM-based Baseline Parser:** We implemented a simple baseline that uses an LLM (GPT-4.1 API) to generate an AST in JSON format directly. We provide the same AST definition as used by SQLFlex. We use a rule-based printer to convert the AST into a query to prevent hallucinations from an LLM-based pretty-printer and align with SQLFlex also using a rule-based printer.

Metrics. Evaluating the semantic correctness of ASTs in a multi-dialect setting is inherently challenging for two reasons. First, different parsers adopt different AST representations, which can lead to structurally different, yet semantically equivalent ASTs for the same query. For example, `pglast` represents both the `CAST` function and the `::` operator using a unified type-casting node. Similarly, SQLGlot represents both `IF` and `CASE` expressions using a single expression type. Second, constructing reliable ground-truth ASTs is challenging due to the inaccessibility of official reference parsers; specifically, while some DBMSs offer official standalone parser packages, for example `pglast`, which is derived directly from PostgreSQL's source code, many others only embed their parsers within the database engine (*e.g.,* DuckDB). Extracting such embedded parsers would incur significant engineering effort.

To address the challenge in evaluating SQL parsers, we adopt property-based testing [10], a technique that checks whether a system behaves as expected according to specific properties. In particular, we used a widely used round-trip property to measure parser correctness [13, 65, 81], and satisfying such a property is useful for developing query rewriting tools [6]. The round-trip property checks whether a query can be parsed into an AST and printed back to the same query as the input, offering a practical way to evaluate parsers without needing ground-truth labels.

We introduce *Query Round-trip (Q-RT) Rate*, the metric for our evaluation. Q-RT is an extension of the round-trip property, which mitigates inaccurate evaluation results caused by normalization applied by baselines. Given an initial query Q_0, we define intermediate parsing and printing steps

as follows:

$$Q_1 = \text{Print}(\text{Parse}(Q_0)), \quad Q_2 = \text{Print}(\text{Parse}(Q_1)) \tag{1}$$

Based on these steps, we define the Q-RT property, which checks that the queries remain identical through each parse-print cycle (*i.e.,* $Q_0 = Q_1 = Q_2$). It provides insights on whether SQLFlex can consistently parse a query into an AST and transform the AST back to an executable query. We relax Q-RT for `pglast` and SQLGlot to only require $Q_1 = Q_2$, to avoid disadvantaging them due to normalization (*e.g.,* `pglast` converts `::` operators into `CAST` functions). Additionally, we ignore whitespace and parentheses during comparison to avoid false alarms from formatting differences. *Q-RT Rate* is calculated as the ratio of the number of queries satisfying Q-RT to the total number of queries in each dataset. Higher values indicate better dialect-agnostic parsing effectiveness and reliability. Note that while the round-trip property cannot reliably detect incorrect ASTs, it serves as a necessary condition for practical parser correctness. We excluded SQLess from this comparison, as its implementation guarantees round-trip equivalence, but produces ASTs with limited practical utility, as user-defined rules often fail to interpret the AST. The selected baselines satisfy the round-trip property while also producing usable ASTs, making them suitable for this evaluation.

In addition to Q-RT, we further evaluate the parsing correctness of SQLFlex by assessing semantic correctness, where `pglast` and the PostgreSQL queries naturally serve as the ground-truth. Following a methodology of previous work on evaluating correctness of DDL statement parsing [17], we extract a core set of semantic elements from a ground-truth AST and compare whether these elements match those extracted from the ASTs produced by SQLFlex and SQLGlot. Specifically, at the clause level, we extract column names from the `SELECT` clause, table names from the `FROM` clause, aliases and their targets, as well as joined table names and join types from the `JOIN` clauses. At the expression level, we extract operators, their literal operands, and their precedence. Since different parsers may normalize certain operators differently, we restrict extraction to a predefined set of common operators to avoid counting semantically equivalent forms as mismatches. For each semantic element, we compute the percentage of ASTs whose extracted results overlap with the ground truth. We refer to this complementary metric as the *Semantic Match (SM) Rate*. Note that the SM rate also remains a best-effort correctness metric, but it provides a correctness reference, complementing the self-referencing round-trip metric.

Dialect-agnostic effectiveness (Q1). Table 4 presents the results comparing SQLFlex against baselines in Q-RT rate, and reports the geometric mean [18] in the last row. For each dialect, the SQLFlex results are averaged over three runs, with standard deviations below 0.6%, indicating consistent performance. Overall, the results show that SQLFlex outperforms all baselines. `pglast` achieves an expected perfect score for the PostgreSQL dataset, but across all dialects, it only has a geometric mean of 65.94% Q-RT. The standard mode of SQLGlot (SQLGlot_s) performs slightly better, with a geometric mean of 74.43%. The dialect-specific mode (SQLGlot_d) achieves a higher geometric mean of 93.26%, but only on dialects it explicitly supports. This improved performance is due to its manually implemented parsing rules for those dialects. However, it requires substantial human effort and still fails to capture some unique features. The LLM-based baseline parser, although requiring no manual effort, performs poorly at generating consistent ASTs (46.48% geometric mean Q-RT). In contrast, SQLFlex outperforms all baselines with a 96.37% geometric mean Q-RT, and ranges from 91.55% to 100% Q-RT across all eight dialects.

We inspected all failed cases. We find that for `pglast` and SQLGlot, almost all failures are due to parsing errors caused by dialect-specific features. For the LLM-based baseline, failures occur due to inconsistent end-to-end generation that produces different results across the AST

Table 4. Query Round-Trip Rates. Bold indicates the best score for each dataset (excluding `pglast` on PostgreSQL)

Dataset	LLM	`pglast`	SQLGlot$_s$	SQLGlot$_d$	SQLFlex
DuckDB	51.53%	72.42%	92.97%	93.88%	**95.86%**
PostgreSQL	49.30%	100.00%	86.67%	87.99%	**96.88%**
MySQL	66.41%	76.42%	86.91%	97.40%	**99.39%**
ClickHouse	32.27%	47.00%	68.61%	94.00%	**94.67%**
H2	56.15%	81.01%	94.69%	–	**97.86%**
CockroachDB	43.46%	69.98%	70.93%	–	**95.06%**
Cassandra	60.00%	74.00%	76.00%	–	**100.00%**
SurrealDB	27.35%	32.76%	38.46%	–	**91.55%**
Mean	46.48%	65.94%	74.43%	93.26%	**96.37%**

Table 5. SM rates on the PostgreSQL dataset. Bold indicates the best. (Opt.: Operator, Opn.: Operand, Prec.: Precedence)

Parser	Tables	Cols	Alias	Joins	Opt.	Opn.	Prec.
SQLFlex	**94.86%**	**98.09%**	**95.86%**	**97.93%**	**97.68%**	**95.44%**	**94.20%**
SQLGlot$_d$	79.37%	88.40%	85.17%	86.33%	86.58%	84.18%	84.34%
SQLGlot$_s$	77.46%	86.58%	83.26%	84.51%	84.76%	82.35%	82.52%

generation and printing stages. For SQLFlex, the few Q-RT failures are mostly caused by limitations in handling dialect-specific features in the rule-based pretty-printer. A typical example comes from CockroachDB, which allows explicit index annotations like "`table@idx`" [11] in table references. In certain queries in the dataset, this is written as "`cb@w`". Since the generic identifiers (`cb` and `w`) provide no semantic information, the segmenter may incorrectly interpret `cb@w` as a schema-qualified table name. After pretty-printing, the output becomes `cb.w`, which fails to match the original input. Similarly, in the SurrealDB dataset, clauses like "`GROUP ALL`" are incorrectly pretty-printed as "`GROUP BY ALL`". Lastly, a small portion of AST produced by SQLFlex (0–2.5%) contained `Unsegmented` nodes, which we counted as failed cases. These typically arise from LLM misinterpretations in complex queries (*e.g.,* multiple subqueries) and fail to be repaired.

Semantic match rates (Q1). We report the SM rates on PostgreSQL queries in Table 5. Overall, SQLFlex achieves the highest SM rates across all evaluated semantic elements, exceeding 94% at both the clause level and the expression level, and outperforming both variants of SQLGlot. While the overall trend of SM is consistent with Q-RT, SM additionally captures certain semantic mismatches that Q-RT fails to detect. For example, SQLFlex incorrectly interpreted the expression `int2 '2'` in the `SELECT` clause as a column reference, whereas it was actually an implicit type-conversion expression that casts `'2'` to type `int2`. This mismatch was missed by Q-RT, because the printed columns were identical despite the differing semantics. Although such cases can be challenging even for LLM-based parsing, their rarity is reflected in the high SM rates, indicating that SQLFlex remains effective in practice.

7.2 Q2: Ablation Study

In this section, we investigate the contributions of important components of SQLFlex.

Methodology. We compared four variants of SQLFlex to assess the impact of its key components:

Table 6. Ablation study results in Q-RT rates. Bold indicates the best score for each dataset.

Dataset	SQL-92	SQLFlex$_v$	SQLFlex$_a$	SQLFlex$_m$	SQLFlex
DuckDB	31.98%	88.91%	86.42%	90.68%	**95.86%**
PostgreSQL	36.54%	90.89%	85.25%	95.44%	**96.88%**
MySQL	62.57%	95.00%	93.65%	96.92%	**99.39%**
ClickHouse	37.23%	90.38%	90.70%	92.31%	**94.67%**
H2	54.19%	93.30%	91.62%	96.09%	**97.86%**
CockroachDB	34.34%	72.61%	86.71%	85.17%	**95.06%**
Cassandra	53.34%	100.00%	100.00%	98.67%	**100.00%**
SurrealDB	17.47%	82.05%	90.88%	86.34%	**91.55%**
Mean	38.39%	88.77%	90.55%	92.58%	**96.37%**

- **SQL-92:** We removed the entire LLM-based segmenter to demonstrate the effectiveness of segmentation overall.
- **SQLFlex$_v$:** We removed the validation and repair component to examine its contribution to parsing correctness.
- **SQLFlex$_a$:** We removed anchor-based expression segmentation, letting the segmenter process an entire expression.
- **SQLFlex$_m$:** We replaced GPT-4.1 with GPT-4.1-mini as the base model to evaluate whether SQLFlex remains effective with a more lightweight LLM.

We evaluated all variants on eight datasets using the Q-RT rate and also report the number of LLM calls for expression segmentation in SQLFlex and SQLFlex$_a$.

Component contributions (Q2). We compare the geometric mean across the eight datasets with results shown in Table 6. Removing the entire LLM-based segmenter (SQL-92) results in a drastic performance drop to 38.39% Q-RT, highlighting that LLM-based segmentation is highly effective when the grammar-based parser fails to parse dialect-specific features. Removing only the validation and repair module (SQLFlex$_v$) leads to a 7.60% Q-RT drop, showing that while the core segmentation is already reliable, validation further improves robustness by correcting edge cases. Removing only the anchor-based segmentation (SQLFlex$_a$) causes a 5.83% Q-RT drop, indicating that our approach improves expression handling effectiveness. Moreover, SQLFlex$_a$ uses 36.61% more LLM calls in total, showing that the anchor-based strategy improves efficiency by avoiding redundant segmentation of known operators. In queries where the expressions are more complex, we believe the anchor-based strategy would be even more effective and efficient in parsing those expressions. Finally, SQLFlex$_m$, which replaces GPT-4.1 with GPT-4.1-mini, achieves a mean Q-RT of 92.58%, only 3.79% below SQLFlex, showing that SQLFlex remains accurate even for lightweight models.

7.3 Q3: Efficiency

In this section, we investigate SQLFlex's efficiency in query parsing and the overhead of LLM calls.

Methodology. We assessed the performance of SQLFlex on each dataset by measuring parsing time and the number of LLM calls (inclusive of repair attempts). Since the base grammar can affect the average efficiency (*i.e.,* more grammar rule matches indicate fewer LLM calls), we use two grammar configurations to provide a more comprehensive view on the efficiency of SQLFlex:

- **SQL-92:** The standard implementation of SQLFlex.
- **SQL:2003** A newer SQL standard to SQL-92, which extended SQL-92 by adding more features such as window functions.

Table 7. Efficiency metrics for query parsing with SQLFlex. Time is measured in seconds.

Dataset	SQL-92			SQL:2003		
	T_{avg}	T_{LLM}	N_{LLM}	T_{avg}	T_{LLM}	N_{LLM}
DuckDB	6.09	8.96	10101	5.26	9.08	7922
PostgreSQL	6.81	10.74	5422	5.55	10.20	4276
MySQL	1.94	5.13	1930	1.87	5.52	1721
ClickHouse	4.38	7.02	6912	3.76	6.32	6148
H2	2.93	6.35	795	1.97	5.31	624
CockroachDB	6.21	9.44	6610	4.72	7.97	5150
Cassandra	1.29	2.70	142	1.36	2.99	124
SurrealDB	3.99	4.97	1311	2.96	3.78	1047
Mean	3.67	6.39	2274	3.06	5.94	1870 (↓21.6%)

For each configuration, we report the average parsing time (T_{avg}) and total number of LLM calls (N_{LLM}) per dialect. We also measure the average parsing time specifically for queries that required segmentation (T_{LLM}).

Efficiency (Q3). We present the results in Table 7. Under the standard SQL-92 configuration, each query takes on average 3.67 seconds to parse, while queries requiring segmentation take 6.39 seconds. Given the limited coverage of the base grammar and the prevalence of dialect-specific features in the collected queries, frequent LLM invocations are expected (as shown in Section 7.2), leading to increased parsing overhead. After extending support to SQL:2003 features, we observe a 21.6% reduction in the number of LLM calls and a 16.62% decrease in average parsing time. This improvement can be attributed to the inclusion of additional grammar rules that capture commonly used SQL features. Moreover, with more anchors introduced in SQL:2003, such as in window functions, segmentation itself achieves a further 7.04% speed-up. Under the SQL:2003 configuration, queries that required segmentation incurred a median of 1 to 6 LLM calls across the eight dialects. The worst case that we encountered was an artificially constructed PostgreSQL query containing 90 type-cast operators `::` to an unknown `inet` type, which triggered 94 LLM calls. Note that this worst case is an outlier rather than typical behavior, and the number of LLM calls is highly workload-specific. In practice, users can extend the grammar with frequently used SQL features, such as the `::` operator to further reduce parsing overhead, while still having the flexibility to handle queries beyond the defined grammar.

7.4 Q4: Parsing DDLs

In this section, we explore whether SQLFlex can be extended beyond query parsing to support Data Definition Language (DDL) statements. As highlighted by the SchemaPile [17] work, which collected database schemas, parsing DDL statements in a multi-dialect setting also remains a major challenge. Among DDL statements, `CREATE TABLE` statements are particularly difficult to parse due to the large number of dialect-specific features, including diverse data types and DBMS-specific keywords, as observed by SchemaPile [17]. Consequently, we focus our evaluation on the `CREATE TABLE` DDL type.

Extending SQLFlex. Extending SQLFlex to support `CREATE TABLE` statements primarily required engineering effort, while using the same hybrid segmentation algorithm. Specifically, this extension involved adding the corresponding SQL-92 grammar rules, AST node definitions, segmentation prompts, and mappings between segmented components and AST nodes. We illustrate this extension with a simplified example below, with dialect-specific features in red. Segmentation proceeds in two iterations. In the first iteration, SQLFlex identifies the table name (`t`) and the column definitions,

Table 8. Correctness of parsers on CREATE TABLE statements. Bold indicates the best. (PK: Primary Key, FK: Foreign Key)

Parser	Tables	Cols	NotNull	Unique	PK	FK
SQLFlex	**99.75%**	97.38%	**99.46%**	**99.72%**	**99.80%**	96.41%
SQLGlot$_d$	97.41%	**97.64%**	88.50%	89.03%	97.64%	**97.64%**
SQLGlot$_s$	90.86%	91.10%	82.29%	83.21%	91.10%	91.10%

while treating the Engine option as dialect-specific. In the second iteration, SQLFlex segments the column definition into data type (mediumint), column name (id), and column constraints (NOT NULL).

```
CREATE TABLE t (id mediumint NOT NULL) Engine=InnoDB;
```

Methodology. We followed the methodology of SchemaPile [17] to assess the correctness of parsing results. Specifically, we used pglast as the reference parser and treated its parse results as the ground truth, using the CREATE TABLE statements from the SchemaPile dataset. After excluding statements that failed to be parsed by pglast and applying keyword-based deduplication, 18,139 CREATE TABLE statements in the PostgreSQL dialect remained. From the parsed ASTs, we extracted and compared the following elements: table names with identifier delimiters stripped, column names appearing in column definitions, and the counts of NOT NULL, UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints. For each element, we calculated the percentage of ASTs whose extracted results match the ground truth.

DDL-parsing effectiveness (Q4). We present the results in Table 8. SQLFlex achieves consistently high correctness across all elements, exceeding 96.41% in every category, and outperforms SQLGlot on four out of the six elements, demonstrating the applicability of SQLFlex to parsing DDL statements. We observed that CREATE TABLE statements have a generally high success rate, partly because they are less challenging than SELECT statements, which tend to contain more dialect-specific features. Although SQLFlex performs slightly worse than SQLGlot$_d$ on column definitions and foreign key constraints, the difference is only at about 1%. Moreover, when considering the percentage of queries for which all elements are parsed correctly, SQLFlex has a substantially higher correctness rate of 93.79%, compared to 80.69% for SQLGlot$_d$.

8 Discussion and Limitation

Effectiveness. To further improve the effectiveness of SQLFlex, several directions can be explored. First, retrieval-augmented generation may help resolve ambiguities during segmentation by retrieving relevant documentation from the target SQL dialect [89]. Second, fine-tuning the LLM could improve its hierarchical reasoning capabilities [32]. However, this approach is less scalable and presents challenges in curating representative datasets across diverse dialects [64]. Third, although prompt engineering was not a primary focus of our work, more refined prompting strategies may further improve segmentation effectiveness. Lastly, as LLMs continue to advance, we expect the effectiveness of SQLFlex to grow accordingly. Given that current LLMs are autoregressive, we believe the sequential nature of our approach will remain relevant.

Efficiency. While the parsing efficiency of SQLFlex does not yet match that of rule-based parsers, it remains suitable for non-interactive use cases or scenarios involving repeated operations on the same query, such as integrating SQL linting into CI/CD pipelines. The main efficiency bottleneck lies in the large number of LLM calls required. We expect the efficiency of SQLFlex to improve, as various techniques to accelerate LLM inference continue to be proposed [23, 36, 63]. Such advancements are also anticipated by other LLM-powered interactive systems, such as Text-to-SQL

tools [78, 79]. Beyond model-level optimization, SQLFlex can further reduce the number of LLM calls by extending the base grammar with additional grammar rules for common SQL features, thus avoiding redundant segmentations. For instance, when inspecting queries that required LLM-based segmentation, we found that some common, but non-standard features like the `::` type-casting operator and `LIMIT` clause were often contained. By adding such features to the grammar, the performance could be further optimized in practice. In an additional experiment, where we added the `::` and `LIMIT` features to the SQL:2003 grammar, we could further reduce the number of LLM calls by 36.33% on the PostgreSQL dataset, for example. Moreover, existing grammar inference techniques that automatically derive context-free grammars from parse trees [27, 70] could enable SQLFlex to cache recurring query patterns or dynamically integrate newly learned rules into its grammar-based parser, further minimizing the need for LLM invocations.

Soundness. Unlike traditional parsing algorithms that provide formal correctness guarantees, SQLFlex adopts a best-effort hybrid parsing approach due to its usage of LLMs. However, achieving full semantic correctness is inherently difficult in a multi-dialect setting without explicit dialect knowledge, and this challenge also affects traditional multi-dialect parsers such as SQLGlot. A representative example is the `||` operator: in MySQL it denotes logical OR, while in PostgreSQL and SQLite it performs string concatenation. Moreover, PostgreSQL assigns `||` lower precedence than arithmetic operators, whereas SQLite assigns it higher precedence. As a result, the expression `3 * 2 || '7'` evaluates differently across the three DBMSs (`true`, 81, and 67 for MySQL, SQLite, and PostgreSQL, respectively). Notably, SQLGlot assigns `||` the lowest precedence uniformly across dialects, which is incorrect for SQLite. As shown in our evaluation, traditional parsers such as SQLGlot also suffer from parsing and correctness issues due to the diversity and evolution of SQL dialects. In contrast, SQLFlex achieves high correctness on the PostgreSQL dataset and proves effective in practical applications such as SQL linting and test-case reduction, where constructing a structurally sound AST across diverse dialects is more important than strict semantic guarantees. Finally, SQLFlex's validation and repair mechanisms further improve its robustness in practice.

Use cases. While this paper reports the results of applying SQLFlex to only two applications, SQLFlex is, in principle, applicable to any AST-based SQL analysis or rewriting workflow rather than being limited to task-specific scenarios. Additional applicable use cases include SQL-level optimization (*e.g.,* Calcite faces similar dialect-related parsing challenges [34]), data lineage and provenance analysis [53, 54], and evaluations of Text-to-SQL systems [40, 86]. Some applications, such as SQL-level optimization, are correctness-critical. While SQLFlex lacks formal correctness guarantees, it can be paired with SQL solvers [31] to strengthen semantic assurances, as demonstrated by recent agentic query optimization frameworks that use LLMs to generate optimized queries [74]. Since SQL solvers also face challenges in multi-dialect support, further research on making them dialect-aware might be needed. The strong practical demand for cross-dialect parsing is further reflected in the significant engineering effort behind open-source projects, such as SQLGlot, SQLFluff, and Calcite, whose issue trackers [24, 25, 34] indicate that dialect coverage remains a persistent pain point. Yet, despite this clear demand, prior research has offered few broadly applicable solutions. SQLFlex aims to fill this gap by providing a dialect-agnostic parsing framework that benefits SQL tooling beyond the use cases presented.

9 Related Work

SQL parsers. Existing SQL parsers can be categorized into generated parsers and hand-written parsers. Generated parsers are automatically built from formal grammar specifications. For example, ANTLR [4] generates parsers that use the LL(*) algorithm [61]. Some DBMSs, such as PostgreSQL, use YACC-style toolkits [51] for their built-in parsers. Mühleisen et al. proposed

using Parsing Expression Grammars [19] as a more extensible and efficient parser generator for DuckDB [56]. Hand-written parsers offer greater flexibility for dialect-specific syntax. A state-of-the-art example is SQLGlot [80], a widely used multi-dialect SQL parser with over 8,000 GitHub stars, supporting more than 30 dialects. However, such tools require substantial manual effort to maintain and extend. `sqlparse` [1] is a non-validating parser that builds approximate parse trees, but is unreliable for query rewriting [15]. To the best of our knowledge, SQLFlex is the first work that integrates LLM with grammar-based parsing to achieve dialect-agnostic query parsing.

SQL dialects. Multiple approaches have addressed the SQL dialect problem in various applications. In DBMS testing, tools like Sedar [21] and QTRAN [48] leverage LLMs to translate test cases across different dialects. Recent Text-to-SQL benchmarks like MiniDev [41] (extending to MySQL and PostgreSQL) and Spider 2.0 [39] (including BigQuery and Snowflake) now incorporate more dialects in addition to SQLite. Closely related are tools like SQL-GEN [64], which synthesizes dialect-specific training data, and new architectures like MOMQ's Mixture-of-Experts model [49] and Exec-SQL's use of execution feedback [85], all aimed at improving multi-dialect capabilities of LLMs. Dialect translation is crucial for applications like data migration. CrackSQL [91] is a state-of-the-art tool combining rule-based and LLM-based approaches for translation. In contrast to these works that focus on generating or translating queries between dialects, SQLFlex addresses the foundational challenge of dialect-agnostic SQL parsing.

Query rewriting. Many applications can be categorized as query rewriting under our formulation. Beyond the discussed use cases, widely researched rewriting tasks include query optimization and DBMS testing. In query optimization, predefined rules are applied to improve query performance [28]. More recent approaches, such as WeTune [83] and LearnedRewrite [92], automatically discover and apply rewrite rules. LLM-R^2 [43] extends this by using LLMs to explore more effective rewrite rules. QueryBooster [7] introduces a domain-specific language for user-defined optimization rules. In DBMS testing, numerous works rewrite queries to detect bugs. Mutation-based fuzzers like WingFuzz [45] and LEGO [44] mutate queries to generate test inputs. Some approaches, like EET [33], AMOEBA [50], and SQLancer [66, 67], generate equivalent queries to check for consistent results. Commonly, these approaches rely on parsers to analyze and modify query ASTs. SQLFlex addresses a key limitation by enabling dialect-agnostic query parsing, broadening the applicability of query rewriting tools.

10 Conclusion

Existing SQL parsers often require significant manual effort to support diverse SQL dialects, limiting the applicability of query analysis and rewriting tools. In this paper, we have presented SQLFlex, a dialect-agnostic query rewriting framework. Our key idea is to integrate grammar-based parsing with LLM-based segmentation for dialect-agnostic query parsing. We introduced clause-level and expression-level segmentation to decompose the hierarchical structure of queries into sequential tasks. We also proposed validation methods to improve reliability. Our evaluation has shown that SQLFlex is practical in real-world SQL linting and test-case reduction tasks. Additionally, SQLFlex can parse 91.55% to 100% of the queries across eight dialects, outperforming baselines. We believe SQLFlex can be applied to a broad range of applications, and requires minimal manual adaptation for dialect-specific features.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions. This project is supported by the Ministry of Education, Singapore, under the Academic Research Fund Tier 1 (FY2023).

References

[1] andialbrecht. 2024. sqlparse. https://github.com/andialbrecht/sqlparse.
[2] Andreas Seltenreich. 2022. SQLsmith. https://github.com/anse1/sqlsmith.
[3] ANSI/ISO. 2017. SQL-92. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.
[4] ANTLR. [n. d.]. ANTLR. https://www.antlr.org/.
[5] Apache. 2018. Apache Calcite. https://calcite.apache.org/.
[6] Apache. 2025. datafusion-sqlparser-rs. https://github.com/apache/datafusion-sqlparser-rs.
[7] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 2911–2924. doi:10.14778/3611479.3611497
[8] Bikash Chandra, Ananyo Banerjee, Udbhas Hazra, Mathew Joseph, and S. Sudarshan. 2019. Automated Grading of SQL Queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1630–1633. doi:10.1109/ICDE.2019.00159
[9] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal* (2015), 731–755. doi:10.1007/s00778-015-0395-0
[10] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. 268–279. doi:10.1145/351240.351266
[11] CockroachDB. [n. d.]. Select from a specific index. https://www.cockroachlabs.com/docs/v25.2/select-clause#select-from-a-specific-index.
[12] Keith Cooper and Linda Torczon. 2003. *Engineering a Compiler: International Student Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[13] Nils Anders Danielsson. 2013. Correct-by-construction pretty-printing. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-Typed Programming (DTP '13)*. doi:10.1145/2502409.2502410
[14] DB-Engines. 2024. DB-Engines Ranking. https://db-engines.com/en/ranking.
[15] Visweswara Sai Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. 2020. SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2331–2345. doi:10.1145/3318464.3389754
[16] DuckDB. [n. d.]. Expressions. https://duckdb.org/docs/stable/sql/expressions/overview.
[17] Döhmen et al. 2024. SchemaPile: A Large Collection of Relational Database Schemas. *Proc. ACM Manag. Data* (2024).
[18] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* (1986). doi:10.1145/5666.5673
[19] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. doi:10.1145/964001.964011
[20] Markus Frohmann, Igor Sterner, Ivan Vulić, Benjamin Minixhofer, and Markus Schedl. 2024. Segment Any Text: A Universal Approach for Robust, Efficient and Adaptable Sentence Segmentation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. doi:10.18653/v1/2024.emnlp-main.665
[21] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3639210
[22] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proceedings of the VLDB Endowment* (2024), 1132–1145. doi:10.14778/3641204.3641221
[23] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of Machine Learning and Systems*. 325–338.
[24] Github Issue. 2025. Add Dremio dialect #4904. https://github.com/tobymao/sqlglot/issues/4904.
[25] Github Issue. 2025. Add H2 dialect #6763. https://github.com/sqlfluff/sqlfluff/issues/6763.
[26] Github Issue. 2025. [MySQL] Unable to handle VALUES ROW() #4371. https://github.com/antlr/grammars-v4/issues/4371.
[27] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. doi:10.1145/3368089.3409679
[28] Goetz Graefe and David J DeWitt. 1987. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. 160–172.
[29] Moshe Hazoom, Vibhor Malik, and Ben Bogin. 2021. Text-to-SQL in the Wild: A Naturally-Occurring Dataset Based on Stack Exchange Data. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty (Eds.). 77–87. doi:10.18653/v1/2021.nlp4prog-1.9

[30] Yuan He, Zhangdie Yuan, Jiaoyan Chen, and Ian Horrocks. 2024. Language Models as Hierarchy Encoders. In *Advances in Neural Information Processing Systems*. 14690–14711.

[31] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proc. ACM Program. Lang.* OOPSLA1 (2024). doi:10.1145/3649849

[32] Zhuohang Jiang, Pangjing Wu, Ziran Liang, Peter Q. Chen, Xu Yuan, Ye Jia, Jiancheng Tu, Chen Li, Peter H. F. Ng, and Qing Li. 2025. HiBench: Benchmarking LLMs Capability on Hierarchical Structure Reasoning. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25)*. 11 pages. doi:10.1145/3711896.3737378

[33] 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)*. USENIX Association, Santa Clara, CA, 821–835.

[34] Jira Issue. 2023. Support Doris Dialect. https://issues.apache.org/jira/browse/CALCITE-5725.

[35] Bill Karwin. 2010. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf.

[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[37] Louisa Lambrecht, Tim Findling, Samuel Heid, Marcel Knüdeler, and Torsten Grust. 2025. Democratize MATCH_RECOGNIZE! *Proc. VLDB Endow.* (2025). doi:10.14778/3750601.3750644

[38] LangChain. [n. d.]. LangChain Documentation. https://python.langchain.com/docs/introduction/.

[39] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763* (2024).

[40] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proceedings of the VLDB Endowment* (2024), 3318–3331. doi:10.14778/3681954.3682003

[41] 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 C C 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 *37th Conference on Neural Information Processing Systems (NeurIPS 2023)*.

[42] Jianling Li, Meishan Zhang, Peiming Guo, Min Zhang, and Yue Zhang. 2023. LLM-enhanced Self-training for Cross-domain Constituency Parsing. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. doi:10.18653/v1/2023.emnlp-main.508

[43] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R^2 : A Large Language Model Enhanced Rule-Based Rewrite System for Boosting Query Efficiency. *Proceedings of the VLDB Endowment* (2024). doi:10.14778/3696435.3696440

[44] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing . In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. doi:10.1109/ICDE55515.2023.00057

[45] Jie Liang, Zhiyong Wu, Jingzhou Fu, Yiyuan Bai, Qiang Zhang, and Yu Jiang. 2024. WingFuzz: Implementing Continuous Fuzzing for DBMSs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[46] 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.

[47] 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. doi:10.1145/3650212.3680317

[48] 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.*, Article ISSTA033 (June 2025). doi:10.1145/3728908

[49] Zhisheng Lin, Yifu Liu, Zhiling Luo, Jinyang Gao, and Yu Li. 2024. MoMQ: Mixture-of-Experts Enhances Multi-Dialect Query Generation across Relational and Non-Relational Databases. arXiv:2410.18406 [cs.CL] https://arxiv.org/abs/2410.18406

[50] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*. doi:10.1145/3510003.3510093

[51] Tony Mason and Doug Brown. 1990. *Lex & yacc*. O'Reilly & Associates, Inc., USA.

[52] mattmc3. [n. d.]. Modern SQL Style Guide. https://gist.github.com/mattmc3/38a85e6a4ca1093816c08d4815fbebfb.

[53] Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day, Sudeepa Roy, and Jun Yang. 2020. I-Rex: an interactive relational query explainer for SQL. *Proc. VLDB Endow.* (2020). doi:10.14778/3415478.3415528

[54] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. doi:10.1145/3299869.3319866

[55] Microsoft. 2025. Transact-SQL reference. https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16.

[56] Hannes Mühleisen and Mark Raasveldt. 2025. Runtime-Extensible Parsers. In *15th Conference on Innovative Data Systems Research, CIDR 2025*.

[57] Ananjan Nandi, Christopher D Manning, and Shikhar Murty. 2025. Sneaking Syntax into Transformer Language Models with Tree Regularization. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*.

[58] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language. In *CIDR*.

[59] OpenAI. [n. d.]. GPT-4.1. https://platform.openai.com/docs/models/gpt-4.1.

[60] Rongqi Pan, Taher A. Ghaleb, and Lionel Briand. 2023. ATM: Black-Box Test Case Minimization Based on Test Code Similarity and Evolutionary Search. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. 12 pages. doi:10.1109/ICSE48619.2023.00146

[61] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. doi:10.1145/1993498.1993548

[62] pglast. 2024. https://github.com/lelit/pglast.

[63] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems*. 606–624.

[64] Mohammadreza Pourreza, Ruoxi Sun, Hailong Li, Lesly Miculicich, Tomas Pfister, and Sercan O. Arik. 2024. SQL-GEN: Bridging the Dialect Gap for Text-to-SQL Via Synthetic Data And Model Merging. arXiv:2408.12733 [cs]

[65] Tillmann Rendel and Klaus Ostermann. 2010. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. doi:10.1145/1863523.1863525

[66] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States) *(ESEC/FSE 2020)*. doi:10.1145/3368089.3409710

[67] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020). doi:10.1145/3428279

[68] Stuart Russel and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach, 4th edition*.

[69] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* (2025).

[70] Michael Schröder and Jürgen Cito. 2022. Grammars for free: toward grammar inference for Ad Hoc parsers. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '22)*. doi:10.1145/3510455.3512787

[71] Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. 2018. Smelly relations: measuring and understanding database schema quality. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. 55–64. doi:10.1145/3183519.3183529

[72] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. doi:10.1145/2786805.2786878

[73] Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John Morcos, Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. 2024. SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL. *Proc. VLDB Endow.* (2024). doi:10.14778/3685800.3685826

[74] Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. 2025. QUITE: A Query Rewrite System Beyond Rules with LLM Agents. arXiv:2506.07675 [cs.DB] https://arxiv.org/abs/2506.07675

[75] SQLancer. 2020. SQLancer. https://github.com/sqlancer/sqlancer.

[76] SQLFluff. 2025. SQLFluff. https://docs.sqlfluff.com/en/stable/index.html.

[77] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371. doi:10.1145/3180155.3180236

[78] Yuan Tian, Jonathan K. Kummerfeld, Toby Jia-Jun Li, and Tianyi Zhang. 2024. SQLucid: Grounding Natural Language Database Queries with Interactive Explanations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. doi:10.1145/3654777.3676368

[79] Yuan Tian, Zheng Zhang, Zheng Ning, Toby Jia-Jun Li, Jonathan K. Kummerfeld, and Tianyi Zhang. 2023. Interactive Text-to-SQL Generation via Editable Step-by-Step Explanations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. doi:10.18653/v1/2023.emnlp-main.1004

[80] tobymao. 2025. SQLGlot. https://github.com/tobymao/sqlglot.

[81] Marcell van Geest and Wouter Swierstra. 2017. Generic packet descriptions: verified parsing and pretty printing of low-level data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2017)*. doi:10.1145/3122975.3122979

[82] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* (2024). doi:10.14778/3681954.3682024

[83] 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 *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 94–107. doi:10.1145/3514221.3526125

[84] Webpage. 2023. The SQL Standard – ISO/IEC 9075:2023 (ANSI X3.135). https://www.iso.org/standard/76583.html.

[85] Jipeng Zhang, Haolin Yang, Kehao Miao, Ruiyuan Zhang, Renjie Pi, Jiahui Gao, and Xiaofang Zhou. 2025. ExeSQL: Self-Taught Text-to-SQL Models with Execution-Driven Bootstrapping for SQL Dialects. arXiv:2505.17231 [cs.CL] https://arxiv.org/abs/2505.17231

[86] Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic Evaluation for Text-to-SQL with Distilled Test Suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.18653/v1/2020.emnlp-main.29

[87] Suyang Zhong and Manuel Rigger. 2024. Understanding and Reusing Test Suites Across Database Systems. *Proc. ACM Manag. Data* (2024). doi:10.1145/3698829

[88] Suyang Zhong and Manuel Rigger. 2025. Scaling Automated Database System Testing. arXiv:2503.21424 https://arxiv.org/abs/2503.21424

[89] Suyang Zhong and Manuel Rigger. 2025. Testing Database Systems with Large Language Model Synthesized Fragments. https://arxiv.org/abs/2505.02012

[90] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2021. SIA: Optimizing Queries Using Learned Predicates. In *Proceedings of the 2021 International Conference on Management of Data*. doi:10.1145/3448016.3457262

[91] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Transaltion System. *Proc. ACM Manag. Data* (2025).

[92] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System Using Monte Carlo Tree Search. *Proceedings of the VLDB Endowment* (2021), 46–58. doi:10.14778/3485450.3485456

[93] Feng Zhu, Lijie Xu, Gang Ma, Shuping Ji, Jie Wang, Gang Wang, Hongyi Zhang, Kun Wan, Mingming Wang, Xingchao Zhang, Yuming Wang, and Jingpin Li. 2022. An Empirical Study on Quality Issues of eBay's Big Data SQL Analytics Platform. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 33–42. doi:10.1145/3510457.3513034

Received October 2025; revised January 2026; accepted February 2026