SQLMorph: Query Mutation and Fine-Grained Metrics for Text-to-SQL Evaluation

Mohammadhossein Malekpour, Mohamed Riahi, Maxime Lamothe, Amine Mhedhbi
Polytechnique Montréal
{mohammadhossein.malekpour, mohamed.riahi, maxime.lamothe, amine.mhedhbi}@polymtl.ca

***Abstract*—Text-to-SQL systems translate natural language queries into executable SQL, democratizing access to structured data. Despite recent advances driven by large language models (LLMs), evaluation remains a major bottleneck: public benchmarks fail to capture the complexity of enterprise schema, while building private evaluation sets is costly and nondeterministic, making evaluation results difficult to reproduce. To address this issue, we present SQLMorph, a framework for Text-to-SQL evaluation via query mutation. SQLMorph introduces two techniques to automatically generate and expand evaluation sets: Join Query Expansion (JQE), which systematically increases structural complexity through valid join additions, and Textual Query Augmentation (TQA), which generates controlled natural language perturbations to assess robustness to linguistic variation. JQE and TQA create targeted choke points to challenge specific system components. When applied to state-of-the-art systems, JQE increases query coverage and reveals accuracy degradation as the number of joins grows. Meanwhile, TQA shows that linguistic brittleness induced by heavy abbreviation can reduce accuracy by up to 17%.**

Beyond evaluation sets, SQLMorph introduces a family of execution-level metrics that address the limitations of current binary measures, such as Execution Accuracy. We define Execution Precision (EXP) and Execution Recall (EXR) to quantify the fraction of correct and recovered results, respectively, and combine them via F1 for unified scoring. Our experiments show that these relaxed metrics enable fine-grained analysis of over- and under-prediction, revealing differences across systems that binary metrics obscure. Together, SQLMorph's query mutation and fine-grained metrics support debugging and better align Text-to-SQL evaluation practices with real-world deployments.

***Index Terms*—Text-to-SQL, Evaluation, SQL, Natural Language, Metrics, Expansion, Augmentation.**

I. Introduction

Natural language interfaces to databases aspire to democratize data access. They allow users to express information needs in natural language (NL) and obtain answers directly from structured data. At the core of this vision lies the *Text-to-SQL* task, which translates NL queries into executable SQL. While the task has witnessed remarkable progress [1], the advent of large language models (LLMs) has substantially improved translation accuracy and spurred the development of new techniques. This momentum has, in turn, driven the rapid emergence of benchmarks such as Spider [2] and BIRD [3], and enabled early, controlled production deployments.

Despite this progress, current evaluation practices remain far from capturing the complexity of real-world enterprise workloads. Enterprise queries often span multiple tables, exhibit intricate join structures, and refer to schema elements with domain-specific terminology and abbreviations. In contrast, public benchmarks typically feature short, structurally simple queries defined over intuitive schema names. This leads to an overly optimistic assessment of system capabilities and thereby distorts cross-system comparisons. Recent efforts such as the Beaver benchmark [4] seek to mitigate these limitations by introducing more complex schemas. Nevertheless, progress remains incremental and labor-intensive, advancing only through the development of new benchmarks one at a time.

Evaluating Text-to-SQL systems on private datasets provides a more realistic alternative. Ultimately, the choice of a production Text-to-SQL system should depend on its performance on the target workload. However, constructing high-quality evaluation sets over private data is expensive, requiring substantial manual annotation, domain expertise, and increasingly, LLM-based automation. These processes are nondeterministic and difficult to control, making reliable and reproducible evaluation challenging. As a result, evaluation in both public and private settings remains fragmented, costly, and hard to scale.

Evaluation methodology is further constrained by the metrics used to measure system performance. The dominant standard, *Execution Accuracy* (EX), assigns a score of 1 when the predicted query's output relation exactly matches that of the ground-truth query and 0 otherwise. While intuitive, this binary measure collapses a wide spectrum of outcomes into a single judgment, failing to capture partial correctness or provide diagnostic feedback. A query that retrieves nearly all correct tuples but misses one condition is indistinguishable from one that returns an irrelevant result. Similarly, structural differences such as column reordering or the inclusion of irrelevant columns are penalized equally. Consequently, EX provides only a coarse, outcome-based view of correctness. This coarseness leaves evaluation misaligned with real-world requirements, where understanding the degree to which a system errs and identifying which components of a Text-to-SQL system succeed or fail are essential.

These challenges stem from a common limitation: current evaluation practices remain static, relying on fixed benchmark queries in the form of NL–SQL pairs and on coarse binary metrics. To address these challenges, we reconceptualize Text-to-SQL evaluation as a dynamic process and introduce *SQLMorph*, a new evaluation framework. *SQLMorph* automatically generates valid query variants to create *choke points* and stress

test system components, and introduces new execution-level metrics that move beyond binary correctness. These metrics are fine-grained and intended for offline development and evaluation to support diagnosis of over- and under-prediction, where ground-truth queries and outputs are available.

A. Contributions

- **Join Query Expansion (JQE)** (Section §IV)
  We introduce *Join Query Expansion* (JQE), an algorithmic technique for increasing SQL complexity by expanding the set of joined tables in a query. JQE performs semantic validation to ensure that each expansion introduces genuine additional reasoning complexity, for example, by requiring joins that cannot be inferred transitively. It further applies diversity-aware pruning to maximize structural coverage across the generated variants. The resulting queries remain executable while becoming substantially more challenging. On BIRD, JQE doubles the average join degree and increases join cyclicity by an order of magnitude, leading to an Execution Accuracy (EX) drop of up to 20% for state-of-the-art systems. Overall, JQE stresses the SQL generation component along the dimension of join-intensive query construction.

- **Textual Query Augmentation (TQA)** (Section §V)
  We propose *Textual Query Augmentation* (TQA), a method for modifying schema and natural-language naming to evaluate robustness to textual perturbations. TQA systematically transforms schema elements or NL queries into less natural, abbreviation-heavy forms (*e.g.*, *WaterTemperature* $\rightarrow$ *WtTp*) while preserving semantics and executability. The resulting variants remain semantically faithful yet expose substantial sensitivity to naming naturalness, with an Execution Accuracy (EX) drop of up to 17% across systems. TQA therefore probes linguistic robustness and context understanding, both of which are central to practical Text-to-SQL deployment.

- **Fine-Grained Execution Metrics** (Section §VI)
  We introduce two fine-grained execution-level metrics, Execution Precision (EXP) and Execution Recall (EXR), to complement the standard Execution Accuracy (EX). EXP measures the fraction of predicted tuples that are correct, while EXR measures the fraction of ground-truth tuples that are recovered. Their harmonic mean, F1, provides a unified summary, while the individual metrics expose over-prediction and under-prediction separately. Together, these metrics provide a more diagnostic view of system behaviour and reveal differences between systems that EX alone obscures.

These contributions establish a framework for *evaluation set construction through query mutation and fine-grained metrics*. To support reproducibility, we release the SQLMorph framework and all experimental scripts as open-source software.[1]

[1]https://github.com/dais-polymtl/sql-morph

II. Background

Modern Text-to-SQL systems typically rely on a multi-stage pipeline [5], [6], [7], [8], [9], as illustrated in Fig. 1. A central design principle of *SQLMorph* is to create targeted *choke points* that systematically challenge specific components of this pipeline. Understanding these components is therefore essential to interpreting our evaluation results.

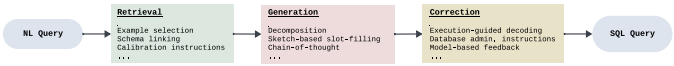

Fig. 1. Text-to-SQL pipeline.

1) Retrieval: This stage collects the contextual information needed to translate an NL query into SQL. This context may include schema elements such as tables, columns, and data types, representative database values, external documentation such as column descriptions, query-writing instructions, and demonstration examples [10], [11], [12]. Retrieval methods are commonly divided into two categories: *filtering*, which narrows the schema and candidate values to those most relevant to the query, and *augmentation*, which enriches the prompt with additional useful context. Typical augmentation operations include entity extraction and the retrieval of explanatory passages through vector- or keyword-based search [13], [14].

2) Generation: Given the NL query and retrieved context, the generation stage has two main substeps: it first produces one or more candidate SQL queries and then selects a final output. Candidate generation methods include few-shot prompting, intermediate representations such as NatSQL [15], plan decomposition for complex queries, and sampling strategies that vary prompt templates or decoding temperature [16], [17], [18]. The selection step then identifies the final SQL query using techniques such as self-consistency voting, rule-based filtering, and reranking. In practice, generation and selection are often interleaved in an iterative loop that runs for up to k rounds, where feedback from the selection step guides the next round of candidate generation [19].

3) Correction: The correction stage aims to repair remaining errors in the selected SQL query. Common techniques include *execution-based feedback*, which executes queries to identify issues such as syntax errors or empty results and then revises them accordingly, and *model-based feedback*, which relies on auxiliary critic models or unit-test-style checks to detect and fix errors. Many systems apply correction iteratively, progressively refining plausible candidates into executable and semantically correct SQL queries [20].

Some systems include additional stages to refine generation through user feedback [21] or to reduce cost via LLM routing [22]. SQLMorph is designed to selectively target such stages through query-based mutations.

III. Experimental Setup

This section outlines the experimental setup used to evaluate SQLMorph in Sections IV–VI. All of our experiments aim

to reflect a realistic, medium-token-budget setting, with an explicit emphasis on minimizing cost for both generation and potential integration into CI/CD pipelines. We view our parameter defaults as a cost-conscious choice rather than a universally optimal setting, since the appropriate trade-off depends on the user's evaluation goals and resources.

1) Dataset: Our experiments use the BIRD benchmark [3], a large-scale cross-domain dataset that has become a de facto standard for Text-to-SQL evaluation. BIRD contains 12,751 NL–SQL pairs across 95 databases spanning 37 domains. Each NL query may include *evidence*, *i.e.*, external text that explains schema attributes, values, or computations, offering richer context for queries within specific domains.

The dataset is divided into three splits: training (9,428 queries over 69 databases), development (dev) (1,534 queries over 11 databases), and hidden test (1,789 queries over 15 databases). We conduct all of our analyses on the dev set.

2) Models: SQLMorph includes a model manager component that supports a range of backends, including OpenAI, Hugging Face, and Ollama, enabling users to pin model versions and, when desired, to run local open-source models for reproducibility and to control model drift. However, for the experiments in this paper, unless otherwise specified, we use `GPT-4o` and `text-embedding-3-small` from OpenAI as our models of choice.

3) Metrics: For now, we adopt EX as the primary evaluation metric. EX compares the output relation of a predicted query $\hat{R}_i$ with that of the ground-truth query R_i, assigning a score of 1 for an exact match and 0 otherwise. The EX score on N queries is computed as follows:

$$EX = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}(R_i = \hat{R}_i) \text{ where } \mathbf{1}(R_i, \hat{R}_i) = \begin{cases} 1 & \text{if } R_i = \hat{R}_i \\ 0 & \text{otherwise.} \end{cases}$$

While this metric is standard, later sections (§VI) introduce SQLMorph's fine-grained alternatives that are relaxed to capture partial correctness and aid in error diagnosis.

4) Text-to-SQL Systems: We consider three representative open-source Text-to-SQL systems that differ in architecture as well as their retrieval and correction strategies. Table I summarizes their reported performance on the BIRD leaderboard at the time of their submission. They are described below:

1. DIN-SQL [10] performs schema retrieval to align NL mentions with tables, columns, and values, then classifies queries as easy, non-nested complex, or nested complex to adapt prompting strategies. It employs the intermediate representation NatSQL [15] and applies a final self-correction stage via an LLM invocation.
2. MAC-SQL [11] follows the three-stage pipeline from Section II, while adding a *decomposer* to break complex NL queries into subproblems, generate subqueries for each during the generation stage, and then merge them.
3. CHESS [13]. It augments the pipeline with an information retriever (*IR*) that extracts keywords, values, and contextual hints from the NL query and schema. Its candidate generator (*CG*) repairs syntax errors, while a *unit tester (UT)* executes candidates and selects the one passing the most validation checks. We adopt $\text{CHESS}_{\text{IR+CG+UT}}$, the strongest reported configuration.

TABLE I
PERFORMANCE AS EXECUTION ACCURACY (EX) OF THREE OPEN-SOURCE TEXT-TO-SQL SYSTEMS ON THE BIRD BENCHMARK AT THEIR TIME OF SUBMISSION.

System		Dev	Test
Human (upper bound)		–	93.0
$\text{CHESS}_{\text{IR+CG+UT}}$	[13]	68.3	71.2
MAC-SQL	[11]	57.6	59.6
DIN-SQL	[10]	50.7	55.9

This setup allows us to evaluate the three primary contributions of SQLMorph: Join Query Expansion, Textual Query Augmentation, and Fine-Grained Execution Metrics.

IV. JOIN QUERY EXPANSION

A. Overview

Enterprise workloads frequently contain multi-table queries with complex join structures; yet public Text-to-SQL benchmarks underrepresent such patterns. *Join Query Expansion (JQE)* within SQLMorph addresses this gap by deterministically increasing a query's structural complexity through the addition of valid joins.

Given an NL–SQL pair, JQE produces one or more expanded variants by adding a new table and valid join conditions. By iterating n times over an evaluation set, JQE bootstraps queries with up to n additional joins. Thus, applying JQE to an evaluation set can generate harder queries from only a small seed subset.

Intuition (running example): Suppose the original query retrieves customer names and total spending:

```
SELECT c.name, SUM(p.amount)
  FROM Customer c
  JOIN Purchase p ON c.id = p.cid
 GROUP BY c.name
```

If the schema contains `Purchase(pid, cid, sku)`, `Product(sku, price, category)`, and `Customer(id, ...)`, JQE may expand the join query graph (JQG) by adding `Product` via `p.sku = Product.sku`, optionally exposing new attributes (*e.g.*, grouping by `category`) while maintaining the original intent and adding more information to the output. This stresses generation under larger join graphs (more tables, new predicates, and possibly cyclic JQGs) without resorting to free-form LLM generation.

B. Design Principles

Two principles guide JQE's design: (i) *determinism and limited hallucination*: the core expansion primitive is algorithmic and graph-driven rather than generative, and LLMs are used only at the end to regenerate an NL query consistent with the expanded SQL; and (ii) *user control*: users can constrain generation via diversity-aware pruning and preferences over join topology and *coverage*. We define coverage in our join

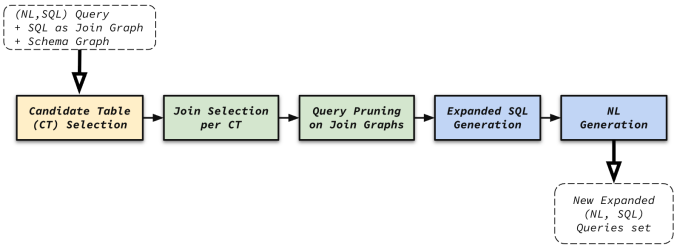

Fig. 2. Multi-stage pipeline for join query expansion.

expansion context as introducing new join patterns, *i.e.*, JQGs that are non-isomorphic to those in an existing evaluation set. Finally, we ensure that each added join is not redundant, *i.e.*, contributes genuine structural complexity. Specifically, the added join must not be implied through transitivity by the existing joins, and must therefore introduce at least one new explicit join condition in the SQL query.

C. Approach

Our approach to JQE is divided into three parts:

1) Input:

- **Schema graph.** A graph $G_S = (V_S, E_S)$ whose vertices are tables and whose edges encode *joinability*. Each edge $e = (t_i, t_j) \in E_S$ carries one or more labels $L(e)$, where each $\ell \in L(e)$ denotes an admissible equi-join predicate (*e.g.*, $t_i.a = t_j.b$) derived from PK–FK constraints or other declared key compatibilities (*e.g.*, FK–FK joins).
- **Seed query.** NL–SQL pair $(Q_{\text{NL}}, Q_{\text{SQL}})$ with Q_{SQL}'s join query graph $G_Q = (V_Q, E_Q)$, where $V_Q \subseteq V_S$ are the tables appearing in `FROM` / `JOIN`, and E_Q are the explicit join predicates.
- **User preferences.** Optional parameters that bound and prioritize (ascending/descending) the expansion process:
  - *Join complexity preference:* whether to favour expansions that increase the number of join conditions per added table, or to restrict expansions to simpler, single-edge joins.
  - *Centrality:* prioritization of expansions to more central tables in G_S (*e.g.*, by degree or betweenness).
  - *Query budget:* the maximum number of expanded NL–SQL pairs to generate.
  - *Pattern coverage:* the maximum number of distinct, non-isomorphic join query graph (JQG) patterns to retain with reference to an input evaluation set.

  By default, we prioritize higher complexity (more joins), arbitrary centrality, do not set a query budget, and set a maximum of a single unique JQG.

2) Output: A set of expanded NL–SQL pairs produced by augmenting the seed query with one table $t_e \in V_S \setminus V_Q$ and one or more admissible join predicates from $L(e)$. Each resulting variant is retained only if it satisfies the semantic and diversity constraints specified by the user preferences. The corresponding NL query is rewritten to reflect the added table and any newly projected attributes, with the aim of preserving the original query intent.

3) Multi-stage pipeline: Fig. 2 illustrates JQE as a multi-stage pipeline, and Algorithm 1 specifies its operations. We briefly describe each stage below.

- **Stage A.** Candidate Table Selection. Using G_S, we enumerate candidate expansion tables, $v_e \in V_S \setminus V_Q$, that are adjacent to at least one $v_t \in V_Q$. Each (v_t, v_e) pair yields a potential join-condition expansion.
- **Stage B.** Join-Condition Combination (JCC) Generation. For each candidate table v_e, let $\{jc_k\}$ be the set of join labels on edges between v_e and $\{v_t \in V_Q\}$. We consider all non-empty subsets (*power set*) of $\{jc_k\}$ as possible expansions for v_e, then prune combinations that introduce *redundant* joins (defined below).
- **Stage C.** Diversity-Aware Pruning. We rank JCCs using user preferences (*e.g.*, by added joins and centrality) and retain up to n queries per *unique* join pattern, where uniqueness is determined via JQG isomorphism (`IS_UNIQUE_JQG` in Alg. 1). This preserves structural variety while controlling cost.
- **Stage D.** Expanded SQL Synthesis. For each retained JCC, we programmatically inject the new table and join predicates into Q_{SQL} (*e.g.*, by extending `FROM`/`JOIN` and `ON`/`WHERE` clauses). Optionally, simple predicates or projections may be added.
- **Stage E.** NL Regeneration. A few-shot LLM prompt rewrites the NL query so that table/column mentions reflect the added join(s), while preserving the original intent. We keep prompts fixed for reproducibility.

In summary, given G_S, JQE (A) identifies candidate tables for expansion, (B) enumerates all non-empty join-condition combinations (JCCs) for each candidate table, (B′) discards JCCs with redundant joins, (C) performs diversity-aware pruning, (D) programmatically synthesizes the expanded SQL query, and (E) regenerates the corresponding NL query via an LLM invocation. We next expand on redundancy checks and diversity-aware pruning.

D. Semantic Redundancy Check

We remove redundant joins from the JQE-generated results. A join redundancy occurs when a candidate join is *implied transitively* by existing joins (*e.g.*, $T_1.A = T_2.A$ and $T_2.A = T_3.A$ imply $T_1.A = T_3.A$). Such joins do not increase complexity and rarely appear in manually authored queries. We detect redundancy through `HAS_REDUNDANCY` in Alg. 1 by examining cycles that include the candidate join expansion set, modeled as edges in the JQG; any JCC that forms such a transitive cycle is discarded.

E. Diversity-Aware Pruning via JQG Isomorphism

We further refine the generated queries by pruning structurally equivalent variants. Two queries are considered *structurally equivalent* when their JQGs are isomorphic [23]. To allow flexibility, user preferences may specify that up to

Algorithm 1 EXPAND_JOINED_TABLES

Input: schema graph G_S; NL–SQL pair $(Q_{\text{NL}}, Q_{\text{SQL}})$; $G_S = (V_S, E_S)$; preferences; `EvalSet`
CTs $\leftarrow \emptyset$ // CTs: candidate tables for expansion
for $v_t \in V_Q$ **do** ▷ Stage A
  for $v_e \in \text{NEIGHBORS}_{G_S}(v_t) \setminus V_Q$ **do**
    CTs $\leftarrow$ CTs $\cup \{(v_t, v_e)\}$
  end for
end for
JCCs $\leftarrow \emptyset$ // join-condition combinations
for $(v_t, v_e) \in$ CTs **do** ▷ Stage B
  $\mathcal{J} \leftarrow \text{JOIN\_LABELS}(v_t, v_e)$
  for $S \in \text{POWERSET}(\mathcal{J}) \setminus \{\emptyset\}$ **do**
    if $\neg\,\text{HAS\_REDUNDANCY}(G_Q, S)$ **then** ▷ Stage C′
      JCCs $\leftarrow$ JCCs $\cup \{(v_e, S)\}$
    end if
  end for
end for
for $(v_e, S) \in \text{RANK}(\text{JCCs}, \text{prefs})$ **do** ▷ Stage C
  $Q'_{\text{SQL}} \leftarrow \text{INJECT\_JOINS}(Q_{\text{SQL}}, v_e, S)$ ▷ Stage D
  if $\text{IS\_UNIQUE\_JQG}(Q'_{\text{SQL}}, \texttt{EvalSet})$ **then**
    $Q'_{\text{NL}} \leftarrow \text{REWRITE\_NL}(Q'_{\text{SQL}})$ ▷ Stage E
    `EvalSet` $\leftarrow$ `EvalSet` $\cup \{(Q'_{\text{NL}}, Q'_{\text{SQL}})\}$
  end if
end for

TABLE II
QUERY COUNT IN THE SETS: BIRD DEV AND JQE OUTPUTS.

	BIRD dev	Generated	Pruned	Expansion
Query Count	1,534	6,873	6,815	58

n queries be retained for each unique JQG, with n=1 by default. We call the retained queries the *expansion set* and the discarded queries the *pruned set*; together, they constitute the *generated set*.

F. Experimental Analysis

To evaluate JQE, we follow the experimental setup in Section III and apply JQE to the BIRD dev set with the default user preferences.

1) JQE Output Structural Distribution: JQE generates a total of 6,873 queries (the *generated set*). Among these, 58 constitute the *expansion set*, consisting of queries that are non-isomorphic to any query in the BIRD dev set or to any previously retained query, while the remaining 6,815 are *pruned* because they are isomorphic to a query in either the BIRD dev set or the expansion set already retained. Overall, the generated set is 4.5× larger than BIRD dev, and all queries execute with non-empty results. Table II provides a summary.

Since JQE rewrites NL queries after SQL expansion (Stage E), we perform a lightweight human verification of NL–SQL pair alignment on the full expansion set (58 pairs). Each expanded NL–SQL pair is labelled as one of the following: *fully captures the intent* (aligned; no edits needed), *simply adds context/noise* (compatible but underspecified for the added join or columns), or *loses the intent* (misaligned; would require correction). We find that 74.1% (43/58) fully capture the intent, 13.8% (8/58) simply add context or noise, and 12.1% (7/58) lose the intent and require correction. This indicates that most rewritten NL queries remain faithful, while a small fraction exhibit underspecification or drift and require human or automated verification.

TABLE III
THE PERCENTAGE OF CYCLIC AND ACYCLIC JOIN QUERY GRAPHS FOR THE ORIGINAL BIRD QUERIES AND THE GENERATED AND EXPANSION SETS.

Set	Acyclic (%)	Cyclic (%)
BIRD dev	99.73	0.27
Generated Set	95.69	4.31
Expansion Set	48.28	51.72

Next, we analyze the resulting structural distribution by comparing the average degrees and cyclicity of the JQGs in the BIRD dev set and the expansion set.

Join Query Graph Node Degree. We measure connectivity using the average node degree of the JQG, $\bar{d} = 2|E|/|V|$. For BIRD dev, $\bar{d}$=0.82; for the generated set, $\bar{d}$=1.35; and for the expansion set, $\bar{d}$=1.80. Higher connectivity is driven by the default preference for greater join complexity, *i.e.*, more join conditions in each expansion.

We further analyze per-query changes in the average node degree by comparing each original JQG (G_{orig}) with its expanded counterpart (G_{exp}). Across the 6,873 generated queries, the most common degree increments are $\Delta\bar{d}$=0.33 (3,910 cases), $\Delta\bar{d}$=0.17 (1,356 cases), and $\Delta\bar{d}$=1.00 (1,166 cases). These distributions mirror the structure of the input queries: 58.7% of the queries in BIRD dev originally join two tables, so adding a third table with a single join yields $\Delta\bar{d}$=0.33. Expansions that connect a new table to the two existing ones introduce higher connectivity, producing $\Delta\bar{d}$=1.00.

Join Query Graph Cyclicity. Cyclic patterns are rare in BIRD dev (0.27%) but increase to 4.31% in the generated set and to 51.72% in the expansion set of 58 output queries (Table III). By maximizing the number of join conditions and thereby introducing cycles, JQE reflects the cyclic structures present in the schema graphs of several BIRD databases.

Takeaway. JQE substantially increases structural diversity: in our experiment, it is able to generate 6,873 queries with 58 new join patterns, where the JQG average degree rises from 0.82 to 1.35, and cyclicity from 0.27% to 4.31%. These effects are even more pronounced in the expansion set of 58 final queries.

2) Accuracy after JQE: We evaluate **CHESS**, **DIN-SQL**, and **MAC-SQL** (from Section III-4) on: (i) BIRD dev set; (ii) Q_{ori}: the 30 original queries that yield the 58 expansions; and (iii) Q_{exp}: the 58 expanded queries. Table IV reports the EX of all three systems on each query set.

TABLE IV
EX OF CHESS, DIN-SQL, AND MAC-SQL ON THE FULL BIRD DEV SET, ON THE 30 ORIGINAL SEED QUERIES THAT PRODUCE JQE EXPANSIONS (Q_{ori}), AND ON THE 58 CORRESPONDING EXPANDED QUERIES (Q_{exp}).

System	EX_{dev}	$\text{EX}_{Q_{ori}}$	$\text{EX}_{Q_{exp}}$
CHESS	64.86%	63.33%	44.83%
DIN-SQL	59.11%	60.00%	32.76%
MAC-SQL	55.90%	40.00%	39.66%

TABLE V
DISTRIBUTION OF CHANGES IN EX (ΔEX) BETWEEN ORIGINAL QUERIES AND THEIR CORRESPONDING EXPANDED QUERIES FOR CHESS, DIN-SQL, AND MAC-SQL.

System	ΔEX=1	ΔEX=-1	ΔEX=0
CHESS	8.62%	18.97%	72.42%
DIN-SQL	5.17%	24.14%	70.69%
MAC-SQL	17.24%	8.62%	74.13%

We define $\Delta\text{EX} = -1$ when a system changes from correct (EX=1) on an original query (Q_{ori}) to incorrect (EX=0) on its expanded version (Q_{exp}), $\Delta\text{EX} = 1$ for the reverse, and $\Delta\text{EX} = 0$ for no change. Table V reports the distribution of ΔEX: degradations ($\Delta\text{EX}=-1$) are more frequent than improvements ($\Delta\text{EX}=1$) for CHESS and DIN-SQL, with DIN-SQL most affected (24.14%), followed by CHESS (18.97%).

3) Join Complexity vs. Performance: From the generated set, we sample four groups by JQG size: $(|V|, |E|) \in \{(2,1),(3,2),(4,3),(5,4)\}$. For each group, we keep the number of conditions and projections fixed, use the same number of queries per group (120, total 480), and balance databases within each condition/projection combination. Figure 3 shows EX changes across groups for the three systems. Accuracy declines monotonically with the number of joins, reaching $\leq 15\%$ at higher join counts.

Manual inspection attributes most failures to join-structure changes; a smaller fraction arise from secondary issues such as missing `DISTINCT`, projection errors, or `GROUP BY` mismatches. We ran an additional control experiment to better isolate the cause of the degradation. We use DIN-SQL, which is the most straightforward system to instrument because of its simple pipeline, and evaluate each seed query together with its corresponding expanded variant. Concretely, we first run DIN-SQL on the expanded NL query and capture the prompt of the generation stage. We then keep this context fixed, replace only the NL query with the original one, and regenerate the SQL. This experiment aims to disentangle the effect of join expansion (more involved query) from the effect of noise in the prompt by keeping the same context.

We report the transition rates among cases where the expanded NL query fails ($\text{EX}_{\text{exp}}=0$): Table VI shows what fraction of these failures recover when we swap to the original NL under the same context, *i.e.*, $\Pr(\text{EX}_{\text{org}}=1 \mid \text{EX}_{\text{exp}}=0)$. This recovery rate is substantial for small join sizes (*e.g.*, 54.90% at $n_{\text{join}}=1$) and decreases as joins increase (down to 24.05% at $n_{\text{join}}=4$), indicating that once the context is built for higher-join expansions, even the original NL becomes harder to solve under that fixed context. Despite this decreasing recovery rate, the consistent gains from expanded→original across all groups still show that the additional join reasoning required by the expanded queries is a driver of the observed degradation. Even when degradation is possibly attributable to increased context noise, it is a consequence of queries that require more joins and thus expand to larger schema and evidence.

TABLE VI
DIN-SQL RECOVERY UNDER FIXED EXPANDED-QUERY CONTEXT. FOR CASES WHERE THE EXPANDED QUERY IS INCORRECT ($\text{EX}_{\text{EXP}}=0$), WE REPLACE THE EXPANDED NL QUERY WITH THE ORIGINAL NL QUERY WHILE KEEPING THE GENERATION-STAGE CONTEXT UNCHANGED, AND REPORT THE EX TRANSITION RATES BY JOIN COUNT n_{JOIN}.

n_{join}	$0\to1$ **(recover)**	$0\to0$ **(still fail)**	**Total Qs** ($\text{EX}_{\text{exp}}=0$)
1	54.90% (28)	45.10% (23)	51
2	41.54% (27)	58.46% (38)	65
3	34.67% (26)	65.33% (49)	75
4	24.05% (19)	75.95% (60)	79

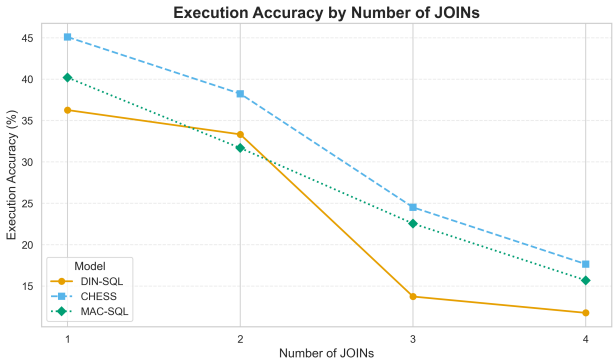

Fig. 3. Execution accuracy (EX) decreases with the number of joins.

Takeaway. JQE efficiently stresses SQL structure: across CHESS, DIN-SQL, and MAC-SQL, EX drops as join complexity increases, confirming that join-heavy queries remain a key failure mode.

V. TEXTUAL QUERY AUGMENTATION

A. Overview

Schema naming conventions exert a strong influence on Text-to-SQL performance. Yet enterprise databases frequently employ domain-specific terminology and abbreviations, whereas public benchmarks predominantly feature more natural names. To alleviate this issue, *Textual Query Augmentation (TQA)* targets the *robustness* of Text-to-SQL systems by systematically altering the natural language (NL) query and schema identifiers in evaluation sets while preserving semantics and executability.

Within *SQLMorph*, TQA serves as a controlled mechanism for testing the retrieval stage, which is known to be sensitive to naming conventions and lexical overlap. Prior studies establish

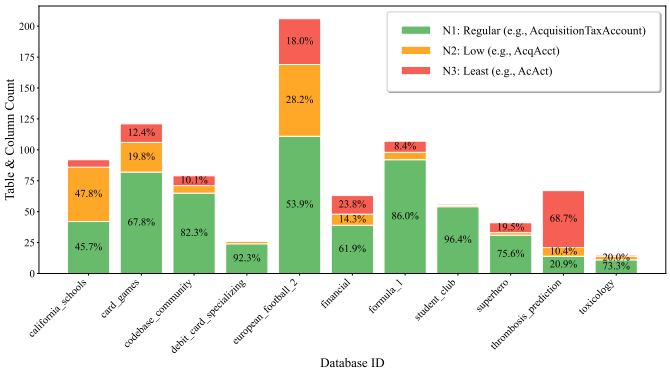

Fig. 4. Distribution of naming naturalness across BIRD dev databases.

an inverse relationship between naming *naturalness* and model accuracy, categorizing identifiers into three levels: regular (N1), low (N2), and least (N3) [24]. TQA challenges the retrieval stage by automatically transforming benchmark data into less-natural variants that maintain the same semantics.

Figure 4 illustrates the distribution of naming naturalness across the BIRD dev set, in which roughly 80% of schema identifiers have natural (N1) forms. This imbalance motivates the need for augmentation techniques that can generate realistic, enterprise-like naming conditions to challenge retrieval stages and human-in-the-loop modules.

B. Design Principles

One of TQA's design principles is to ensure that the observed performance changes arise solely from textual perturbations and not from semantic variation. As such, TQA transforms schema elements and their references within queries and evidence while keeping the SQL executable and replacing an element name in the SQL only when necessary. Values, constants, and clauses are kept fixed to guarantee minimal changes. This ensures that only lexical aspects of the query and not logic or intent, are modified.

C. Approach

Our approach to TQA is divided into three parts:

1) Input:

- **Seed query.** NL–SQL pair with accompanying evidence and relevant schema.
- **Naturalness classifier.** We use the SNAILS classifier [24], which was fine-tuned on `google/canine-s`, to assign each instance to one of three naturalness levels: regular (N1), low (N2), or least (N3).
- **User preferences (optional).** De-naturalization aggressiveness (N1→N3 vs. N1→N2), and target references location: NL, schema, or both. By default, we apply the most aggressive setting, targeting N3 in both the NL query and the schema.

TABLE VII
PERCENTAGE CHANGE IN EX (ΔEX) ON THE BIRD DEV SET UNDER THE THREE TQA NATURALNESS SETTINGS FOR CHESS, MAC-SQL, AND DIN-SQL, RELATIVE TO THE ORIGINAL/ORIGINAL (O/O)

	ΔEX (%)		
NL/Schema	**CHESS**	**MAC-SQL**	**DIN-SQL**
L/O	−7.0	−11.1	−9.2
O/L	−6.7	−9.7	−23.4
L/L	−8.8	−11.7	−23.9

2) Output: A set of minimally changed, *less-natural* references that remain semantically faithful and executable, potentially yielding (i) modified schemas (via `ALTER`); (ii) updated ground-truth SQL queries (with rewritten identifiers); and (iii) updated NL queries and evidence (with only schema mentions rewritten). Each item passes execution equivalence checks (EX = 1 relative to the original query).

3) Technique: TQA executes as follows:

- **Classify Naturalness.** Extract table and column identifiers from the database schema and label each with the SNAILS classifier as N1, N2, or N3, together with a confidence score.
- **Decrease Naturalness.** Transform N1 and N2 identifiers into N3, or into the user-specified target. Using an LLM invocation with temperature 0 and a fixed seed yields transformations such as *WaterTemperature*→*WtTp*.
- **Alter Schema.** Clone each database and apply table and column renames via SQL `ALTER`. Validate the rename was successful (*e.g.*, using `PRAGMA table_info()`).
- **Alter SQL Query.** Update the ground-truth SQL query by replacing the old identifier using regex while avoiding SQL keywords, functions, and values. We discard original NL queries for which the system did not obtain EX = 1.
- **Alter NL Query.** Rewrite the NL query and evidence by substituting only schema mentions with least-natural forms using a fixed few-shot prompt. We avoid changes to values (numbers, dates, entities). By restricting edits to schema mentions, this substitution preserves the original meaning and keeps the NL query semantically consistent.

D. Experimental Setup and Analysis

To evaluate TQA, we follow the experimental setup in Section III. We evaluate four augmentation settings that rename either the NL side (NL query and evidence), the SQL side (schema definition and SQL query), or both:

- **O/O**: Original NL + Original SQL
- **L/O**: Less-natural NL + Original SQL
- **O/L**: Original NL + Less-natural SQL
- **L/L**: Less-natural NL + Less-natural SQL

We apply TQA to the BIRD dev set (1,534 NL–SQL query pairs). We evaluate TQA at both the system and the stage levels, and we provide a qualitative example.

a) System-level analysis: We consider only queries for which a system successfully generated a correct output, *i.e.*, queries for which O/O yields EX = 1, and then measure

TABLE VIII
SCHEMA RETRIEVAL PERFORMANCE ACROSS NATURALNESS SETTINGS. FPR (%): FALSE POSITIVE RATE (LOWER IS BETTER). SLR (%): SCHEMA LINKING RECALL (HIGHER IS BETTER).

	Full-Schema		SCSL		TCSL	
NL/Schema	**FPR**	**SLR**	**FPR**	**SLR**	**FPR**	**SLR**
O/O	90.1	99.5	29.9	15.0	13.7	27.4
L/O	88.0	30.4	29.2	11.7	14.3	21.4
O/L	90.1	96.4	34.1	9.8	16.4	25.4
L/L	88.2	28.2	33.8	14.1	16.1	20.1

changes under (L/O), (O/L), and (L/L). This leads to initial query counts per system as follows: 998 for CHESS, 857 for MAC-SQL, and 904 for DIN-SQL. Table VII reports ΔEX (%) relative to O/O (negative is a drop). All systems degrade consistently. Schema and NL de-naturalization both reduce EX; DIN-SQL shows the largest degradation, followed by MAC-SQL and CHESS. Thus, TQA can increase benchmark difficulty while preserving semantics.

Takeaway. TQA can make queries harder while preserving semantics and executability. We find that regardless of the augmentation setting, O/L, L/O, and L/L, all lead to degradation. For production settings in which the schema cannot change, L/O provides a natural setting to challenge system retrieval. This further highlights the importance of query expansion and data expansion techniques in Text-to-SQL systems.

b) Stage-level analysis: To conduct a stage-level analysis, we adopt the schema retrieval formulation of prior work [25] and consider three retrieval approaches: **Full-Schema** (no filtering; retrieves over the full schema), **TCSL** (Table-then-Column), and **SCSL** (Single-Column). We evaluate them across O/O, O/L, L/O, and L/L on the augmented queries. We report two complementary retrieval metrics: **FPR** (*False Positive Rate*; lower is better) and **SLR** (*Schema Linking Recall*; higher is better). FPR quantifies the proportion of retrieved columns that are irrelevant to the query, thereby measuring retrieval noise. Lower FPR indicates a more precise retriever that introduces less distracting context for downstream generation. SLR, in contrast, measures whether all required columns are retrieved for each query. Results are summarized in Table VIII.

- **L/O.** Relative to O/O, SLR drops sharply for Full-Schema (up to -69.1%) and more moderately for SCSL and TCSL (-3.3% and -6.0%). FPR decreases moderately, with a slight increase for TCSL. These changes point to worse retrieval quality and help explain the EX reduction.
- **O/L.** SLR again decreases: -3.1%, -5.2%, and -2.0% for Full-Schema, SCSL, and TCSL, respectively, while FPR rises for SCSL and TCSL ($+4.2\%$ and $+2.7\%$).
- **L/L.** Moving from L/O to L/L, the additional changes are comparatively small: SLR changes by -6.2%, $+2.4\%$, and -1.3%, accompanied by only slight increases in FPR.

c) Qualitative Example: To illustrate the augmentation process and its downstream effects, we show an example from the `toxicology` database (ID 245) in the BIRD dev set. The original and less-natural versions differ only in schema naming and in its propagation to the NL and SQL, while preserving semantics and execution results. We next present the original and less-natural NL query, the evidence, and the required changes to the relevant schema. We then analyze the behaviour of schema retrieval on this query.

Original NL: *What is the average number of bonds the atoms with the element iodine have?*
Less-natural NL: *What is the average number of bnds the atms with the Elmt iodine have?*

Original Evidence:
atoms with the element iodine refers to element = 'i'; average = DIVIDE(COUNT(bond_id), COUNT(atom_id)) where element = 'i'

Less-natural Evidence:
atms with the Elmt iodine refers to Elmt = 'i'; average = DIVIDE(COUNT(b_id), COUNT(atmId)) where Elmt = 'i'

Original Relevant Schema:
`atom.atom_id`, `atom.element`, `connected.atom_id`, `connected.bond_id`

Less-natural Relevant Schema:
`atm.atmId`, `atm.Elmt`, `conn.atmId`, `conn.b_id`

Schema Retrieval. Under the **O/L** and **L/O** settings, systems frequently fail to retrieve the correct columns, *e.g.*, by missing `atom_id` or `connected.atom_id`, which leads to downstream generation and execution errors.

Takeaway. TQA can stress-test retrieval and query augmentation techniques, as well as system-level query reformulation and understanding. TQA can help identify misalignment failures when matching an NL query to a schema, *e.g.*, in the schema linking stage.

VI. FINE-GRAINED EVALUATION METRICS

A. Motivation

The standard EX metric is binary: the predicted and expected output relations either match exactly or they do not. As a result, it compresses a wide range of outcomes into a single bit and discards potentially important signal. Queries that recover most of the correct rows but miss a single boundary condition receive the same score as queries that return entirely irrelevant results. Likewise, structural differences such as harmless extra columns are penalized as severely as true semantic errors. Consequently, EX obscures partial correctness, limits error diagnosis, and provides little insight into how a Text-to-SQL system fails, whether through under-prediction or over-prediction of rows, columns, or values.

Our proposal: We introduce a family of *fine-grained execution-level* metrics to quantify how a predicted result deviates from the ground truth, rather than only whether it matches exactly. At the core are two complementary measures: *Execution Precision* (EXP), which captures the fraction of predicted cells that are correct, and *Execution Recall* (EXR), which captures the fraction of ground-truth cells that are recovered. Their F1 score provides a compact summary, while separate reporting of EXP and EXR reveals whether errors arise primarily from over-prediction, reflected in lower precision, or under-prediction, reflected in lower recall.

B. Approach

Given a ground-truth query q and a predicted query $\hat{q}$, we execute both to obtain the column-name sets $cols(q), cols(\hat{q})$ and the row multisets $rows(q), rows(\hat{q})$. If $rows(q)$ and $rows(\hat{q})$ are identical over their full schemas, then $\mathrm{EX} = 1$, and we set $\mathrm{EXP} = \mathrm{EXR} = \mathrm{F1} = 1$. Otherwise, we proceed with relaxed matching, which first optionally matches the columns and then the rows and cells. Finally, the evaluator chooses whether to penalize extra predicted columns.

a) Column Matching: We compare only those columns that the evaluator deems matchable; SQLMorph offers three matching regimes:

- **Exact-Column (EC).** Match by exact column name: $c_\cap = cols(q) \cap cols(\hat{q})$.
- **Semantic-Column (SC).** Build a textual descriptor for each column (name plus top-k values), embed the descriptors, restrict to type-compatible pairs and domain values when applicable, and compute a maximum-weight bipartite matching. Keep only pairs above a similarity threshold (*e.g.*, 0.7). For a fixed embedding model, semantic-column matching is fully deterministic and always yields the same matching; changing the model, however, may change the result. Since matching is performed only over output columns (typically fewer than 20), the cubic assignment cost is negligible in practice and independent of the number of output rows.
- **No-Column (NC).** Skip column matching entirely; matching then relies only on row and cell comparisons.

b) Row/Cell Matching: Let $c_\cap$ denote the matched column set. We then match $rows(q)$ and $rows(\hat{q})$ as follows:

- **Exact-Cell (EC).** Under EC or SC, project $rows(q)$ and $rows(\hat{q})$ onto $c_\cap$ and treat the resulting rows as multisets. Under NC, compare rows directly through their implicitly matched value sets similar to the EX computation. For each unique matched row pattern r, let $f_q(r)$ and $f_{\hat{q}}(r)$ denote its multiplicities in $rows(q)$ and $rows(\hat{q})$, respectively. Then

$$|\mathrm{MatchedRows}| = \sum_r \min(f_q(r), f_{\hat{q}}(r)),$$

$$|\mathrm{MatchedCells}| = |\mathrm{MatchedRows}| \cdot |c_\cap|.$$

  Intuition: Credit is assigned only when two rows match exactly over the aligned comparison units, namely matched columns under EC or SC, or implicitly matched values under NC.
- **Partial-Cell (PC).** Under EC or SC, first project $rows(q)$ and $rows(\hat{q})$ onto $c_\cap$; under NC, compare rows through their implicitly matched value sets. Matching then proceeds in two phases:
  - *Phase 1: exact matching.* First, check for exact matches, as in EC. These exact matches are removed, and the remaining unmatched rows are passed to the second phase.
  - *Phase 2: partial matching.* For each remaining $p \in rows(\hat{q})$ and ground-truth $g \in rows(q)$, compute the fraction of equal cells as a similarity score:

$$\mathrm{sim}(p, g) = \frac{\#\{i \in c_\cap : p_i = g_i\}}{|c_\cap|}.$$

    We then greedily select the pair with the highest similarity, add the matching cells $\{i \in c_\cap : p_i = g_i\}$ to PartialMatchedCells, and remove both rows from further consideration. This process continues until all remaining $rows(\hat{q})$ have been considered or until no unmatched $rows(q)$ remain. We sort the rows and columns of the outputs of q and $\hat{q}$ before matching so that equivalent relations for the same named attributes produce deterministic matches.

Finally, the total number of matched cells, $|\mathrm{MatchedCells}|$, is $|\mathrm{ExactMatchedCells}| + |\mathrm{PartialMatchedCells}|$.

c) Accounting for extra predicted columns: We define $|cells(q)| = |rows(q)| \cdot |cols(q)|$ as the total number of ground-truth cells and account for extra predicted columns by choosing one of the following options:

- **Penalize Extras (PE) predicted columns.** $|cells(\hat{q})| = |rows(\hat{q})| \cdot |cols(\hat{q})|$. All predicted columns are counted, so extra columns reduce precision.
- **Ignore Extras (IE) predicted columns.** $|cells(\hat{q})| = |rows(\hat{q})| \cdot |c_\cap|$. Only matched columns are counted, under the assumption that the predicted output already contains the required data and that extra columns may therefore be ignored. This option is unavailable under NC, since no matched column set $c_\cap$ is constructed.

Metrics: The final metrics are defined as follows:

$$\mathrm{EXP} = |\mathrm{MatchedCells}|/|PCells|$$

$$\mathrm{EXR} = |\mathrm{MatchedCells}|/|GCells|$$

$$\mathrm{F1} = 2 \cdot \mathrm{EXP} \cdot \mathrm{EXR}/(\mathrm{EXP} + \mathrm{EXR})$$

Takeaway. EXP is the fraction of predicted cells that are correct, EXR is the fraction of ground-truth cells that are recovered, and $F1$ unifies the two. Depending on the evaluation setting, extra predicted columns may either be penalized or ignored.

TABLE IX
SINGLE-ERROR MUTATIONS AND THEIR EXPECTED EFFECTS ON THE NUMBER OF ROWS (R), COLUMNS (C), VALUES (V), AND EVALUATION METRICS (EXP, EXR). SYMBOLS DENOTE THE EXPECTED DIRECTION OF CHANGE: ↑ INCREASE, ↓ DECREASE, = UNCHANGED, AND × CONTEXT-DEPENDENT.

Operator	Description	(R, C, V)	(EXP, EXR)
projection_drop	Remove one column from the projection list (if > 1 remain).	$(=,\downarrow,=)$	$(=,\downarrow)$
add_star_wildcard	Append * or alias.*/table.* if no star exists.	$(=,\uparrow,=)$	(PE ↓; IE =,=)
distinct_toggle	Remove the DISTINCT keyword.	$(\times,=,=)$	$(\downarrow,\times)$
where_predicate_delete	Remove one predicate from a compound condition.	$(\uparrow,=,=)$	$(\downarrow,\uparrow)$
where_condition_flip	Flip comparison direction ($=\leftrightarrow\neq$, $>\leftrightarrow<$, $\geq\leftrightarrow\leq$).	$(\times,=,=)$	$(\times,\times)$
where_strengthen	Make boundary stricter ($<\rightarrow<=$, $>\rightarrow>=$).	$(\downarrow,=,=)$	$(=,\downarrow)$
where_weaken	Make boundary looser ($<=\rightarrow<$, $>=\rightarrow>$).	$(\uparrow,=,=)$	$(\downarrow,\uparrow)$
where_remove	Remove the entire WHERE clause.	$(\uparrow,=,=)$	$(\downarrow,\uparrow)$
having_condition_flip	Flip aggregate comparison ($=\leftrightarrow\neq$, $>\leftrightarrow<$, $\geq\leftrightarrow\leq$).	$(\times,=,=)$	$(\times,\times)$
having_remove	Remove the HAVING clause.	$(\uparrow,=,=)$	$(\downarrow,\uparrow)$
join_break	Remove the ON condition (cartesian product).	$(\uparrow,=,=)$	$(\downarrow,\times)$
join_type_to_left	Convert non-LEFT joins to LEFT JOIN.	$(\uparrow,=,=)$	$(\downarrow,\uparrow)$
limit_increase	Double the LIMIT and add a random OFFSET.	$(\uparrow,=,=)$	$(\downarrow,\uparrow)$
limit_decrease	Halve the LIMIT (min 1).	$(\downarrow,=,=)$	$(=,\downarrow)$
aggregation_swap	Swap aggregates (SUM↔AVG, MIN↔MAX, COUNT→SUM).	$(=,=,\times)$	$(\downarrow,\downarrow)$

C. Experimental Analysis

1) Controlled Error Sensitivity: Research Question – Can our fine-grained metrics capture and distinguish the effects of isolated single-error mutants in predicted SQL queries?

To evaluate SQLMorph's metrics, we construct a benchmark of single-error SQL mutants by applying each mutation to a ground-truth query and evaluating how the resulting change is reflected in the metrics. Each mutant is created by applying exactly one atomic change to a ground-truth query from the BIRD development set; the resulting mutations are summarized in Table IX. In the mutant-generation code, we enforce depth-1 mutations by exhaustively trying each mutation and retaining only those that both modify the AST and produce syntactically valid SQL. When a query contains subqueries, mutations are applied only to the outer query so that each change remains easy to interpret.

Table IX groups the mutants by SQL component and summarizes their expected effects on rows, columns, values, and our metrics. Some mutants mainly change the number of returned rows. For example, increasing the LIMIT or making a WHERE condition less restrictive tends to add rows, which should lower EXP. In contrast, decreasing the LIMIT or making a WHERE condition more restrictive reduces the rows, which should lower EXR. Other mutants affect columns or values. For instance, dropping a projected column should reduce recall, while adding a wildcard mainly hurts EXP when extra predicted columns are penalized. Mutants that alter aggregate values can reduce both EXP and EXR.

Table X summarizes the results. We report ΔEX, ΔEXP, ΔEXR, and ΔF1. Smaller values below 0 indicate a larger penalty or drop from 1. The results show the main weakness of EX. For every mutant, EX drops to 0%. As a result, it cannot distinguish a mild error from a severe one. In contrast, EXP, EXR, and F1 spread the errors across a much wider range and can reveal how a query is incorrect.

TABLE X
IMPACT OF SINGLE-ERROR MUTATION ON EVALUATION METRICS.

	Evaluation Metrics						
		EC-EC-IE			SC-EC-IE		
Injected Error	ΔEX	ΔEXP	ΔEXR	ΔF1	ΔEXP	ΔEXR	ΔF1
projection_drop	-100	0	-43	-28	0	-43	-28
add_star_wildcard	-100	-93	-10	-87	-82	0	-71
distinct_toggle	-100	-57	-57	-57	-57	-57	-57
where_predicate_delete	-100	-85	-60	-81	-85	-64	-82
where_condition_flip	-100	-96	-61	-94	-96	-61	-94
where_remove	-100	-96	-54	-95	-96	-54	-95
where_strengthen	-100	-65	-60	-63	-65	-60	-63
where_weaken	-100	-65	-75	-73	-65	-75	-73
having_condition_flip	-100	-64	-100	-100	-64	-100	-100
having_remove	-100	-61	0	-51	-61	0	-51
join_break	-100	-98	-42	-98	-98	-46	-98
join_type_to_left	-100	-53	-50	-54	-53	-50	-54
limit_increase	-100	-76	0	-62	-76	0	-62
limit_decrease	-100	-4	-61	-45	0	-60	-43
aggregation_swap	-100	-100	-100	-100	-100	-100	-100

Row-count mutants behave as expected. With limit_increase, the query returns too many rows, so EXP drops sharply while EXR stays at 1. With limit_decrease, the query returns too few rows, so EXR drops while EXP remains nearly unchanged. This shows that EXP and EXR clearly separate over-prediction from under-prediction of rows.

Schema-related mutants also produce clear patterns. projection_drop removes output columns, which lowers EXR while leaving EXP unchanged. add_star_wildcard adds extra predicted columns, which mainly lowers EXP. Under SC matching, this penalty is smaller, suggesting that SC is more robust to harmless schema noise. distinct_toggle lowers both EXP and EXR, showing as expected the effect of duplicates on the multiset comparisons of the results.

Filter and grouping mutants show a range of severity. Mutants such as where_predicate_delete reduce both EXP and EXR. Bigger changes such as removing the WHERE clause or flipping it causes an even larger drop. For grouping-related mutants, removing the HAVING clause behaves like over-selection: EXP decreases while EXR stays high. By contrast, flipping a HAVING condition can drive EXR to zero.

Join mutants have their own effects. `join_break`, which creates a cartesian product, causes one of the largest EXP drops and also lowers EXR. `join_type_to_left` produces a smaller and more balanced degradation, which is consistent with a milder semantic change.

Across mutants, SC usually matches or improves on EC, especially when the error introduces extra but semantically related columns. PC helps mainly when predicted rows are close to the correct ones but not identical, such as after `join_break`; otherwise, its behaviour is similar to EC.

> **Takeaway.** EXP and EXR provide much more useful diagnostic signal than binary EX. They distinguish over-prediction from under-prediction, separate row and column errors, and better capture cases where outputs are partially correct.

2) System-Level Comparison: Research Question – Do granular metrics reveal finer distinctions in how systems fail when compared to EX?

We conducted a comparative analysis of three Text-to-SQL systems, namely, CHESS, DIN-SQL, and MAC-SQL, on the BIRD dev set. Among the $1,534$ total examples, we identified 410 queries for which all three systems produced $EX = 0$, representing $\sim$26.7% of the benchmark. These shared failures pose a particular challenge: EX offers no information for ranking systems or for analyzing their behaviour on failed queries, despite the potential partial correctness of the predictions. Table XI summarizes the results.

The ranking of the systems (CHESS $>$ DIN-SQL $>$ MAC-SQL) remains unchanged under both EX and our fine-grained metrics. However, EXP and EXR reveal meaningful differences within the shared-failure queries: for example, CHESS recovers nearly a quarter of the ground-truth cells, whereas MAC-SQL recovers only about 15%. In addition, precision-recall asymmetries help explain the nature of these failures: CHESS maintains higher EXR, while DIN-SQL exhibits a more balanced trade-off between EXP and EXP. Thus, although the leaderboard order is stable, our metrics add diagnostic resolution. Fig. 5 illustrates this point by showing the distributions of EXP and EXR over failed queries, highlighting how close each system comes to the correct answer on average even when EX assigns them zero.

3) Qualitative Example: Consider the following NL query: 'List the names of schools with more than 30 difference in enrollments between K–12 and ages 5–17. Please also give the full street address of the schools.'

SC-EC-PE Metrics (DIN-SQL vs MAC-SQL).
EX: 0.000 vs 0.000
EXP: 0.393 vs 0.500 (*gap:* 0.11)
EXR: 0.982 vs 1.000 (*gap:* 0.02)
F1: 0.561 vs 0.667 (*gap:* 0.11)

This highlights how large the variation can be among failed queries. Next, we report the predicted and golden SQL queries and provide an explanation.

Ground-truth SQL:

```
SELECT T1.School, T1.Street
  FROM schools AS T1
  JOIN frpm AS T2 ON T1.CDSCode = T2.CDSCode
 WHERE T2.`Enrollment (K-12)` -
     T2.`Enrollment (Ages 5-17)` > 30;
```

DIN-SQL Predicted SQL:

```
SELECT frpm.`School Name`, schools.Street,
    schools.City, schools.Zip, schools.State
  FROM frpm
  JOIN schools
    ON frpm.CDSCode = schools.CDSCode
 WHERE (frpm.`Enrollment (K-12)` -
     frpm.`Enrollment (Ages 5-17)`) > 30;
```

MAC-SQL Predicted SQL:

```
SELECT T2.School, T2.Street, T2.City, T2.Zip
  FROM frpm AS T1
  JOIN schools AS T2 ON T1.CDSCode =
      T2.CDSCode
 WHERE T1.`Enrollment (K-12)` -
     T1.`Enrollment (Ages 5-17)` > 30;
```

Explanation:

- Both systems predict the correct filtering condition for selecting schools with large enrollment differences, but they differ in their projected attributes.
- We would expect both systems to achieve an EXR of 1.0. However, DIN-SQL projects `frpm.'School Name'` instead of `School`, preventing a perfect match.
- DIN-SQL projects five attributes, whereas MAC-SQL projects four, corresponding to three and two extra attributes, respectively. This additional over-prediction lowers their EXP and, consequently, their F1 scores.

> **Takeaway:** This example shows high **EXR** but low **EXP**: both systems recover nearly all expected rows, but are penalized for projecting unnecessary columns. Binary EX treats them as equally wrong, whereas SQLMorph reveals that both are in fact very close to the correct answer.

VII. Related Work and Challenges

We contextualize SQLMorph within recent research and emphasize persistent challenges in Text-to-SQL evaluation.

Existing benchmarks such as *Spider*, *BIRD*, *WikiSQL*, *KaggleDBQA*, and *Beaver* have driven substantial progress in Text-to-SQL research [2], [3], [26], [27], [28], [4]. However, they remain limited in scale, label fidelity, and compositional diversity. Public datasets also tend to emphasize relatively small schema and shallow join structures, which do not fully

TABLE XI

SYSTEM-LEVEL COMPARISON OF CHESS, DIN-SQL, AND MAC-SQL ON THE BIRD DEV SET, RESTRICTED TO THE SHARED-FAILED QUERIES WHERE ALL THREE SYSTEMS HAVE EX = 0. FINE-GRAINED METRICS (EXP, EXR, AND F1) UNDER MULTIPLE MATCHING CONFIGURATIONS REVEAL PARTIAL CORRECTNESS AND DIFFERENCES IN FAILURE BEHAVIOUR.

		Evaluation Metrics (%) – BIRD Shared Failures Subset														
		EC-EC-IE			**EC-PC-IE**			**SC-EC-IE**			**SC-PC-IE**			**NC-PC**		
System	EX	EXP	EXR	F1	EXP	EXR	F1	EXP	EXR	F1	EXP	EXR	F1	EXP	EXR	F1
CHESS	0.00	28.76	19.66	18.44	29.06	19.85	18.82	47.80	16.62	15.14	51.44	19.11	17.82	N/A	N/A	N/A
DIN-SQL	0.00	29.01	21.12	18.42	29.09	21.25	18.51	63.04	19.54	17.18	66.26	22.05	19.71	N/A	N/A	N/A
MAC-SQL	0.00	31.09	16.07	14.63	31.34	16.30	14.84	54.47	13.18	11.91	56.97	15.10	13.82	N/A	N/A	N/A

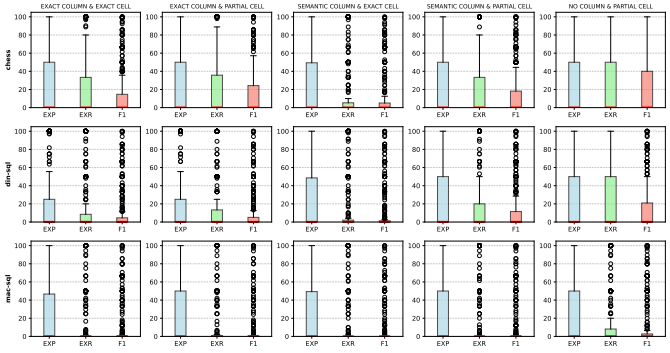

Fig. 5. System-level comparison on the BIRD dev set's common failed queries on combinations of fine-grained metrics.

reflect the complexity of enterprise workloads [29]. Recent studies show that system accuracy drops sharply as join complexity increases [30]. SQLMorph complements these static benchmarks by generating deterministic query variants that increase both structural and lexical diversity. In particular, JQE introduces additional join predicates, thereby stressing systems along an underrepresented dimension in current benchmarks.

A related challenge is that real-world schema differ substantially from those in public datasets. Enterprise databases often contain hundreds of interrelated tables, heterogeneous sources, and cryptic identifiers [4]. Such complexity allows multiple valid SQL queries per NL intent [31], making correctness ambiguous even when intent is preserved [32].

Reproducibility remains another major concern. Progress can be difficult to interpret because results often depend on dataset splits, prompt design, sampling choices, and private evaluation settings [4], [27], [33], [34]. SQLMorph is designed to make evaluation more reproducible by fixing prompts, seeds, and sampling parameters, and by validating generated variants through execution-based checks. It is also suitable for private, in-house evaluation, where schema and workload traces cannot be shared due to privacy, governance, or annotation cost constraints [4], [26]. In this sense, SQLMorph complements recent efforts on enterprise-oriented evaluation such as *Beaver* and *Spider 2.0* [4], [35].

SQLMorph also relates to recent work on schema naming and schema linking. Prior studies have shown that naming quality strongly affects Text-to-SQL performance [24], [26], [36], [25]. For example, *SNAILS* emphasizes improving schema naturalness to facilitate linking [24]. SQLMorph takes a complementary perspective. TQA intentionally denaturalizes schema identifiers and their mentions in natural language to simulate enterprise-style abbreviations and naming conventions. This allows us to stress-test robustness to lexical mismatch, schema linking, and retrieval stages more broadly.

With respect to metrics, most existing leaderboards still rely on binary Execution Accuracy (EX) [2], [3]. Although EX is simple and widely adopted, it collapses all failures into a single bit and does not indicate whether a system under-predicts rows, over-predicts rows, projects extra columns, or otherwise comes close to the correct answer [31], [37], [32]. SQLMorph introduces Execution Precision (EXP) and Execution Recall (EXR), which separate over-prediction from under-prediction and provide more interpretable error diagnostics. SQLMorph adopts relaxed, execution-grounded metrics with cell-level matching, so that semantically equivalent or partially correct predictions can receive partial credit rather than being counted as complete failures. Our metric design also differs from recent proposals based on continuous similarity scores [38], which can be harder to interpret, and may conflate syntactic and semantic differences.

Recent RL-based and reward-driven training methods for Text-to-SQL often rely on proxy, binary, or composite rewards that do not necessarily align with execution correctness at a fine-grained level [39], [40], [41]. These rewards can reduce sparsity or improve alignment with end-task accuracy, but they do not explicitly decompose execution errors into over-prediction and under-prediction in the way that SQLMorph's EXP and EXR do [39], [41]. As such, SQLMorph's metrics may be worth exploring as potential training reward signals.

VIII. CONCLUSION

SQLMorph reframes Text-to-SQL evaluation as a form of controlled stress testing. JQE increases structural complexity through systematic join expansion, while TQA probes robustness to naming variation by de-naturalizing natural-language queries and schema references. Our fine-grained metrics, EXP and EXR, expose over-prediction and under-prediction that binary EX obscures. Together, these components provide a reproducible and diagnostic evaluation framework that better reflects enterprise requirements.

IX. AI-Generated Content Acknowledgement

We used GenAI tools to correct grammar and suggest synonyms or paraphrases for a few paragraphs.

References

[1] A. Quamar, V. Efthymiou, C. Lei, and F. Özcan, "Natural language interfaces to data," *FnT DB*, vol. 11, 2022.

[2] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," *EMNLP*, 2018.

[3] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *NeurIPS*, 2024.

[4] P. B. Chen, F. Wenz, Y. Zhang, D. Yang, J. Choi, N. Tatbul, M. Cafarella, Ç. Demiralp, and M. Stonebraker, "Beaver: an enterprise benchmark for text-to-sql," *CoRR*, vol. abs/2409.02038, 2024.

[5] Z. Hong, Z. Yuan, Q. Zhang, H. Chen, J. Dong, F. Huang, and X. Huang, "Next-generation database interfaces: A survey of llm-based text-to-sql," *CoRR*, vol. abs/2406.08426, 2024.

[6] B. Li, Y. Luo, C. Chai, G. Li, and N. Tang, "The dawn of natural language to sql: Are we fully ready?" *CoRR*, vol. abs/2406.01265, 2024.

[7] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Luo, Y. Zhang, J. Fan, G. Li, and N. Tang, "A survey of nl2sql with large language models: Where are we, and where are we going?" *CoRR*, vol. abs/2408.05109, 2024.

[8] W. Zhang, Y. Wang, Y. Song, V. J. Wei, Y. Tian, Y. Qi, J. H. Chan, R. C.-W. Wong, and H. Yang, "Natural language interfaces for tabular data querying and visualization: A survey," *CoRR*, vol. abs/2310.17894, 2024.

[9] K. Maamari and A. Mhedhbi, "End-to-end text-to-sql generation within an analytics insight engine," *CoRR*, vol. abs/2406.12104, 2024.

[10] M. Pourreza and D. Rafiei, "Din-sql: Decomposed in-context learning of text-to-sql with self-correction," *NeurIPS*, 2023.

[11] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Li, "Mac-sql: A multi-agent collaborative framework for text-to-sql," 2025.

[12] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, and S. O. Arik, "Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql," *CoRR*, 2024.

[13] S. Talaei, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi, "Chess: Contextual harnessing for efficient sql synthesis," *CoRR*, vol. abs/2405.16755, 2024.

[14] X. Xie, G. Xu, L. Zhao, and R. Guo, "Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment," *CoRR*, vol. abs/2502.14913, 2025.

[15] Y. Gan, X. Chen, J. Xie, M. Purver, J. R. Woodward, J. Drake, and Q. Zhang, "Natural sql: Making sql easier to infer from natural language specifications," *EMNLP*, 2021.

[16] Y. D. Dönder, D. Hommel, A. W. Wen-Yi, D. Mimno, and U. E. S. Jo, "Cheaper, better, faster, stronger: Robust text-to-sql without chain-of-thought or fine-tuning," *CoRR*, vol. abs/2505.14174, 2025.

[17] D. Lee, C. Park, J. Kim, and H. Park, "MCS-SQL: leveraging multiple prompts and multiple-choice selection for text-to-sql generation," *COLING*, 2025.

[18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," *NeurIPS*, 2022.

[19] Y. Gao, Y. Liu, X. Li, X. Shi, Y. Zhu, Y. Wang, S. Li, W. Li, Y. Hong, Z. Luo, J. Gao, L. Mou, and Y. Li, "A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql," *CoRR*, vol. abs/2411.08599, 2025.

[20] L. Sheng and S. Xu, "CSC-SQL: corrective self-consistency in text-to-sql via reinforcement learning," *CoRR*, vol. abs/2505.13271, 2025.

[21] K. Maamari, C. Landy, and A. Mhedhbi, "Genedit: Compounding operators and continuous improvement to tackle text-to-sql in the enterprise," 2025.

[22] M. Malekpour, N. Shaheen, F. Khomh, and A. Mhedhbi, "Towards optimizing SQL generation via LLM routing," *CoRR*, vol. abs/2411.04319, 2024.

[23] J. R. Ullmann, "An algorithm for subgraph isomorphism," *JACM*, vol. 23, no. 1, pp. 31–42, 1976.

[24] K. Luoma and A. Kumar, "Snails: Schema naming assessments for improved llm-based sql inference," *SIGMOD*, 2025.

[25] K. Maamari, F. Abubaker, D. Jaroslawicz, and A. Mhedhbi, "The death of schema linking? text-to-sql in the age of well-reasoned language models," *CoRR*, vol. abs/2408.07702, 2024.

[26] B. Qin, B. Hui, L. Wang, M. Yang, J. Li, B. Li, R. Geng, R. Cao, J. Sun, L. Si, F. Huang, and Y. Li, "A survey on text-to-sql parsing: Concepts, methods, and future directions," *CoRR*, 2022.

[27] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *CoRR*, 2017.

[28] C.-H. Lee, O. Polozov, and M. Richardson, "KaggleDBQA: Realistic evaluation of text-to-SQL parsers," Aug. 2021.

[29] A. Mitsopoulou and G. Koutrika, "Analysis of text-to-SQL benchmarks: Limitations, challenges and opportunities," 2025.

[30] T. Eckmann, M. Urban, J.-M. Bodensohn, and C. Binnig, "HLR-SQL: Human-like reasoning for text-to-SQL," 2025.

[31] A. Floratou, F. Psallidas, F. Zhao, S. Deep, G. Hagleither, W. Tan, J. Cahoon, R. Alotaibi, J. Henkel, A. Singla, A. v. Grootel, B. Chow, K. Deng, K. Lin, M. Campos, V. Emani, V. Pandit, V. Shnayder, W. Wang, and C. Curino, "NL2SQL is a solved problem... not!" *CIDR*, 2024.

[32] M. Pourreza and D. Rafiei, "Evaluating cross-domain text-to-SQL models and benchmarks," 2023.

[33] C. Renggli, I. F. Ilyas, and T. Rekatsinas, "Fundamental challenges in evaluating text2sql solutions and detecting their limitations," *CoRR*, vol. abs/242501.18197, 2025.

[34] F. Wenz, O. Bouattour, D. Yang, J. Choi, C. Gregg, N. Tatbul, and Çağatay Demiralp, "Benchpress: A human-in-the-loop annotation system for rapid text-to-sql benchmark curation," *CoRR*, vol. abs/2510.13853, 2025.

[35] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, and T. Yu, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," *CoRR*, vol. abs/2411.07763, 2024.

[36] G. Katsogiannis-Meimarakis and G. Koutrika, "A survey on deep learning approaches for text-to-sql," *VLDBJ*, 2023.

[37] A. Kumar, P. Nagarkar, P. Nalhe, and S. Vijayakumar, "Deep learning driven natural languages text to sql query conversion: A survey," *CoRR*, 2022.

[38] G. Pinna, Y. Perezhohin, L. Manzoni, M. Castelli, and A. De Lorenzo, "Redefining text-to-SQL metrics by incorporating semantic and structural similarity," *Sci. Rep.*, 2025.

[39] M. Pourreza, S. Talaei, R. Sun, X. Wan, H. Li, A. Mirhoseini, A. Saberi, and S. O. Arik, "Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql," *CoRR*, 2025.

[40] Z. Yao, G. Sun, L. Borchmann, Z. Shen, M. Deng, B. Zhai, H. Zhang, A. Li, and Y. He, "Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql," *CoRR*, 2025.

[41] H. Hao, W. Hu, O. Verkholyak, D. A. Tarzanagh, B. Gutow, S. Didari, M. Faraki, H. Moon, and S. Min, "Paverl-sql: Text-to-sql via partial-match rewards and verbal reinforcement learning," *CoRR*, 2025.