LLM-Aided SMT Refutation of SQL Query Equivalence

Jasmine Lesner[1], Fuheng Zhao[1], Xifeng Yan[1], and Amr El Abbadi[1]

University of California, Santa Barbara
{jlesner,fuheng_zhao,xyan,amr}@cs.ucsb.edu

Abstract. Automated optimization of SQL queries hinges on knowing when a rewritten query remains semantically equivalent to the original. Yet many formal verification tools, while rigorous, cannot handle features common in modern SQL (e.g., window functions), leaving large portions of real workloads outside their scope. This paper presents DBDoctor, an computer-aided verification system that uses Large Language Models (LLMs) to rewrite unsupported SQL into verifier-friendly forms and then delegates refutation to a state-of-the-art SMT-based SQL equivalence checker. The verifier's counterexamples are validated against the original queries, creating a self-correcting loop. We show that our LLM aided approach extends verifier coverage to previously unsupported queries, discovers new counterexamples missed under practical time budgets, and remains consistent with the SMT-based verifier where it already succeeds. For database systems, this capability enables safer query optimization, regression testing of query rewrites, and guardrails for automated tuning – impacting reliability and cost at scale – while also exemplifying how LLMs can be combined with formal reasoning.

Keywords: SQL Equivalence Verification · Large Language Models (LLM)· SMT Solvers · Database Systems · Formal Verification · Computer-Aided Verification.

1 Introduction

Our computer aided verification system, called DBDoctor (Fig 1), uses an LLM to rewrite SQL queries containing constructs unsupported by formal SMT verifiers into semantically aligned, verifier-compatible forms, and then delegates the refutation step to SMT verification (e.g., VeriEQL [5]) to produce counterexamples. Crucially, any counterexample found on the rewritten pair is validated against the original queries, forming a self-correcting loop that accepts only counterexamples that show non-equivalence of the original SQL (Fig 2). This design brings several benefits: (i) it broadens the effective domain of formal SQL checkers without changing their core SMT engines, (ii) it preserves rigor by employing SMT reasoning, and (iii) it yields actionable artifacts (counterexample databases) that are useful for debugging query rewrites.

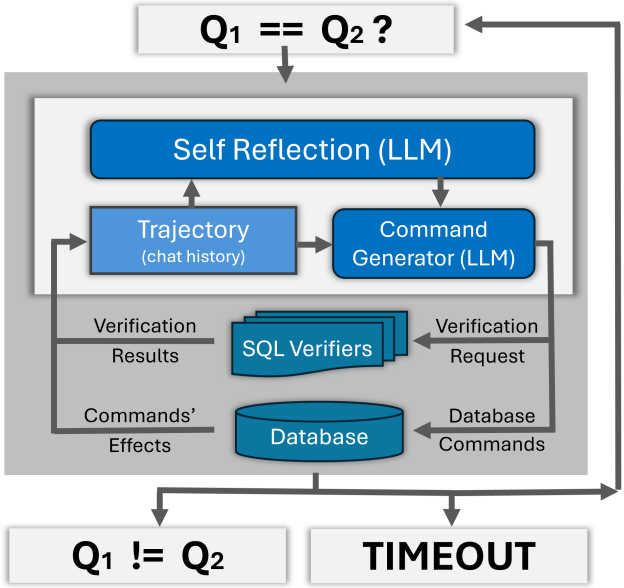

Fig. 1. DBDoctor's workflow is an iterative cycle where the system generates SQL query verification requests and database commands. This continuous feedback loop enables data-driven behavior based on empirical evidence from database tests and SQL verifier results. Our implementation employs a mix of OpenAI's tool-calling GPT-4.1-MINI and O4-MINI 'thinking' models [11].

1.1 Related Work

The intersection of LLMs and databases is a rapidly growing field, especially for text-to-SQL generation and automated optimization [8, 17, 14, 13]. A common theme in optimization is ensuring that a rewritten query is equivalent to the original. Some approaches involve having LLMs select from a predefined set of trusted rewrite rules [9] or use a formal verifier to check LLM-generated optimizations [4]. However, these strategies are limited; a fixed set of rules constrains novelty and may contain bugs, while existing verifiers support only a limited subset of SQL, leaving most complex, real-world queries unverified [5]. The task of verifying SQL equivalence is a well-known, undecidable problem in computer science.

Research has produced two main classes of tools:

1. *Bounded* verification tools like Cosette [2], Qex [15], and VeriEQL [5] support a subset of SQL and aim to prove or disprove equivalence for database instances up to a certain size, producing a concrete counterexample if non-equivalence is found.
2. *Unbounded* verification tools like SPES [19] and HoTTSQL [3] attempt to prove equivalence for all possible database instances but are typically restricted to an even smaller subset of SQL.

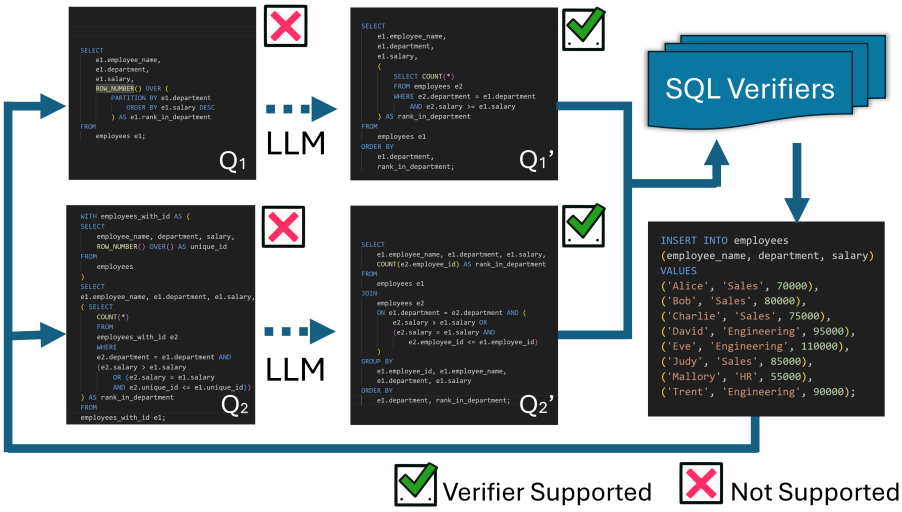

Fig. 2. The core workflow of DBDoctor combines heuristic SQL rewriting by an LLM with formal analysis by an SMT based SQL equivalence verifier. In this example, two queries (Q_1, Q_2) with unsupported window functions are rewritten into equivalent forms (Q_1', Q_2') that use only verifiable SQL constructs. The SQL verifier can then process the rewritten pair to find a counterexample, proving non-equivalence. A crucial final step is to validate this counterexample against the original queries (Q_1, Q_2). If the counterexample is invalid for the original pair, the LLM is prompted to generate a new rewrite, creating a self-correcting loop.

This leaves a significant gap between what can be formally verified and the types of queries used in practice. Recent studies have investigated using LLMs directly for SQL equivalence checking, with prompting techniques like *Miniature & Mull* [18] and providing execution plans as context [12]. While these approaches show LLMs can offer valuable heuristic insights, they also suffer from factual hallucination and instability. Crucially, prior work has not explored an workflow where an LLM can actively use tools or run experiments to validate its hypotheses. Our work builds on these insights by creating a symbiotic system where LLM intuition is disciplined by the rigor of formal methods and grounded by empirical feedback from a live database.

1.2 Contributions

This paper's contributions, evaluated on realistic SQL query pairs, are:

- **A verification-first agentic framework.** We design DBDoctor, which integrates LLM-guided rewriting with a formal SQL verifier and empirical validation, using the verifier as a source of counterexamples.
- **Coverage extension via rewrite-to-verify.** We introduce a method that rewrites complex, unsupported SQL into SMT verifier-compatible forms

without requiring changes to the verifier, thereby expanding its effective coverage.
- **Counterexample-driven soundness checks.** We show that verifier-produced counterexamples on rewritten queries can be validated against the original pair, filtering spurious rewrites and yielding actionable counterexamples of non-equivalence.
- **Practical impact for database tooling.** Our evaluation demonstrates extended coverage on previously unsupported SQL query pairs, discovery of additional counterexamples, and agreement with the SMT verifier where it already succeeds – supporting safer optimizer rule deployment, automated tuning guardrails, and regression testing in practice.

2 Methodology

We formalize the problem as follows. Given two SQL queries Q_1 and Q_2 (often an "optimized" candidate vs. a reference), determine whether $Q_1 \equiv Q_2$ under relational semantics. Let $\mathcal{V}$ be a SQL verifier (e.g., VeriEQL [5]) that accepts only a subset of SQL. Let $\mathcal{S}$ denote that supported SQL subset and let $\mathcal{R}$ be a rewriting procedure driven by an LLM that attempts to map (Q_1, Q_2) to $(Q_1', Q_2') \in \mathcal{S} \times \mathcal{S}$ while *preserving the intended semantics of both queries in lockstep*. Concretely, the LLM is instructed that any structural change introduced to make one side verifiable must be symmetrically applied to the other side to maintain a plausible equivalence relation.

Refutation-by-rewrite loop (Fig 2)

1. **Rewrite.** Use the LLM to propose (Q_1', Q_2') with only constructs in $\mathcal{S}$, conserving schema and inputs.
2. **Verify.** Submit (Q_1', Q_2') to $\mathcal{V}$. If $\mathcal{V}$ produces a bounded counterexample instance I witnessing $Q_1'(I) \neq Q_2'(I)$, proceed; otherwise, adapt the LLM prompt and attempt a new rewrite or return NON-REFUTED.
3. **Validate on originals.** Execute Q_1 and Q_2 on the same I. If $Q_1(I) \neq Q_2(I)$, return REFUTED with counterexample I; else, reject the spurious counterexample and continue the loop.

Soundness for refutation (w.r.t. concrete execution) The system reports REFUTED only when it has validated a counterexample I such that $Q_1(I) \neq Q_2(I)$ on the original queries. Hence, any reported counterexample is a true behavioral discrepancy under standard SQL execution. Completeness is not claimed: if no rewrite lands in $\mathcal{S}$ or the verifier times out, the system may return NON-REFUTED even when the queries differ.

Design choices (i) *Verifier-centric correctness:* the LLM proposes hypotheses; $\mathcal{V}$ and the database executor arbitrate correctness. (ii) *Minimal engineering to extend coverage:* we leverage existing engines (e.g., VeriEQL's SMT procedures) without modifying them, expanding their effective coverage via rewrite-to-verify. (iii) *Actionable artifacts:* validated counterexamples are concrete databases, useful for optimizer debugging, rule regression tests, and CI pipelines in practice.

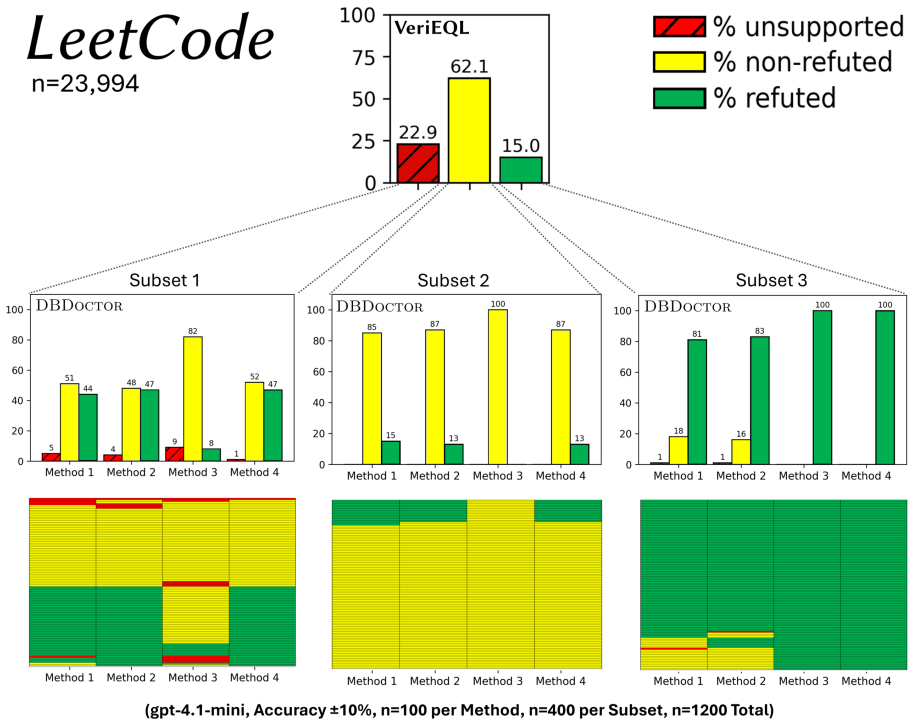

Fig. 3. Performance of DBDoctor vs. VeriEQL, with per-pair detail. *Top Middle:* Running VeriEQL on the full LeetCode set (n=23,994) yields 22.9% unsupported, 62.1% non-refuted, and 15.0% refuted query pairs. These three buckets define our evaluation subsets. *Bottom (2x3 panels):* For each bucket (Subset 1: unsupported; Subset 2: non-refuted; Subset 3: refuted), we sample 100 query pairs and run four methods (Methods 1–4). In each panel, the *upper bar chart* summarizes outcomes for the 100 pairs, while the *heatmap* directly below shows the same results at per-pair resolution: each column is a method, each row is a query pair, and colors match the legend (red = unsupported, yellow = non-refuted, green = refuted). The bar heights are the marginal proportions of colored cells in the heatmap beneath them—i.e., the bar chart is an aggregate view of the per-pair matrix. These results show that DBDoctor expands verifier coverage on previously unsupported pairs (Subset 1), discovers additional counterexamples in non-refuted pairs (Subset 2), and remains consistent with the verifier on refuted pairs (Subset 3).

2.1 Methods Compared

To isolate component contributions we compare:

- **Method 1 (LLM Only):** search for counterexamples without tools.
- **Method 2 (LLM + Database):** executes candidates against a DB to guide counterexample search.
- **Method 3 (LLM + SQL Verifier):** rewrite-to-verify loop using $\mathcal{V}$.
- **Method 4 (LLM + DB + Verifier):** full system with verifier and database validation.

2.2 Benchmark Dataset

We evaluate against VeriEQL [5] using the LeetCode corpus (23,994 pairs) curated in [5], where each candidate is paired with a known-correct reference. Queries are complex and representative of practical patterns; we repair minor scraping errors and adapt to PostgreSQL when necessary.

2.3 Evaluation Protocol

We first run VeriEQL for up to 10 minutes per pair, reproducing [5], and partition into **Unsupported**, **Non-refuted**, and **Refuted**. From each bucket we sample $n = 100$ pairs (margin of error $\approx \pm 10\%$ at 95% confidence) and apply Methods 1–4. We study: (i) *Coverage* (% pairs not UNSUPPORTED); (ii) *Refutation rate*; and (iii) *Agreement* with VeriEQL on the Refuted bucket. All reported refutations include validated concrete counterexamples.

3 Results

Our experimental results are presented in Figure 3. The experiments test four methods across three distinct subsets of query pairs, with each method tested on the same sample of 100 pairs per subset.

Subset 1: Unsupported This tests coverage extension – the central goal. Method 4 reduces UNSUPPORTED from 100% to 1% and refutes 47% of pairs, demonstrating that rewrite-to-verify plus DB validation converts many previously out-of-scope instances into actionable refutations. Method 2 also performs strongly (47% refuted, 4% unsupported), indicating that empirical execution substantially helps the LLM search; Method 3 sometimes fails to produce verifier-acceptable rewrites (9% unsupported), underscoring the benefit of database feedback.

Subset 2: Non-refuted Here we test whether the system can uncover counterexamples that time-bounded VeriEQL missed. Method 1 refutes 15% of pairs, while Methods 2–4 each achieve 13% refutation under the same budget, showing that LLM assistance can expose subtle inequivalences even when a SOTA verifier times out or yields NON-REFUTED. Because every reported counterexample is validated on the originals, these are genuine defects.

Subset 3: Refuted Methods 3 and 4 (those invoking $\mathcal{V}$) achieve 100% agreement – every VeriEQL refutation is reproduced. Methods without the verifier are weaker: Method 2 refutes 83%, Method 1 refutes 81%, highlighting the value of formal reasoning when the problem lies squarely within the verifier's native coverage.

4 Discussion

Application domain and impact Database engines and data platforms increasingly apply automated rewrites – cost-based rules, ML-guided tuning, and human-authored transformations – to control latency and resource cost. A single wrong rewrite can silently corrupt analytics, violate compliance filters, or miscompute business metrics. By furnishing *validated* counterexamples for inequivalent rewrites, DBDoctor provides (i) a guardrail for auto-tuners and LLM assistants, (ii) a regression oracle for optimizer rules, and (iii) a forensic tool for triaging customer-reported inconsistencies. For operators, this translates to lower incident rates and reduced compute spend via safe optimization; for researchers, it demonstrates how LLMs can be used to extend the practical coverage of SMT-based SQL verification.

Why the loop works LLM-driven rewrites need not be perfect; they must only land inside the verifier's coverage while preserving the *relative* transformation across both queries. The verifier excels at *refutation*; when it produces a counterexample, we confirm it on the originals, filtering out artifacts of imperfect LLM rewrites. This division of labor leverages complementary strengths: LLMs to reach $\mathcal{S}$, SMT verification for reasoning, and database execution for final judgment.

Limitations Completeness is bounded by (i) the SMT verifier's scope and timeouts, (ii) the LLM's ability to find a semantics-preserving pair in $\mathcal{S}$, and (iii) nondeterminism and list semantics (Section 5), where simple set-based equality can be misleading. Nevertheless, the refutation results are sound due to database validation.

5 Future Work

List Semantics Current result comparison uses set semantics; outermost `ORDER BY` requires list semantics to avoid false equivalence (Listings 1.1–1.3). Extending validation with stable-order checks – and teaching $\mathcal{R}$ to preserve or align order constraints – would strengthen guarantees.

Listing 1.1. Q_1: A query with a guaranteed ordering.

```
1 SELECT id, name, salary
2 FROM employees
3 WHERE department = 'Engineering'
4 ORDER BY salary DESC;
```

Listing 1.2. Q_2: A query with no guaranteed ordering.

```
SELECT * FROM (
    SELECT id, name, salary
    FROM employees
    WHERE department = 'Engineering'
) AS engineering_employees;
```

Listing 1.3. Q_3: A query where ordering is not guaranteed to be preserved.

```
SELECT * FROM (
    SELECT id, name, salary
    FROM employees
    WHERE department = 'Engineering'
    ORDER BY salary DESC
) AS engineering_employees;
```

Handling Non-Deterministic Queries We plan to model common nondeterminism sources (unordered `LIMIT`, ties in `ORDER BY`, non-deterministic functions, floating-point aggregation; Listings 1.4–1.5) so that counterexamples must be robust across admissible executions, not just a single run.

Listing 1.4. An unordered query with a 'LIMIT' clause is non-deterministic.

```
WITH Employees (EmployeeID, Name, Department) AS (
    SELECT 1, 'Alice', 'Engineering'
    UNION ALL
    SELECT 2, 'Bob', 'Engineering'
)
SELECT EmployeeID, Name, Department
FROM Employees
LIMIT 1;
```

Listing 1.5. Floating-point arithmetic can be non-deterministic.

```
WITH number_set (id, val) AS (
  VALUES
    (1,  1e18::double precision),
    (2, -1e18::double precision),
    (3,  1.0::double precision)
)
SELECT
    -- result_a is 1.0 since the large numbers cancel
    SUM(val ORDER BY id ASC) AS result_a,
    -- result_b is 0.0, since the small number is lost
    SUM(val ORDER BY val DESC) AS result_b,
    -- result_c is unknown due to unspecified  order
    SUM(val) AS result_c
FROM
    number_set;
```

Leveraging the Small-Scope Hypothesis Timeouts in bounded verification suggest exploring counterexample downscaling and upscaling strategies informed by the small-scope hypothesis [10, 5], converting hard instances into tractable ones without losing discriminative power.

Tree-Structured LLM Interaction Rather than a single linear history (Fig. 4), a tree-of-thoughts exploration [16] could expand several rewrite candidates in parallel and prioritize promising branches by verifier feedback.

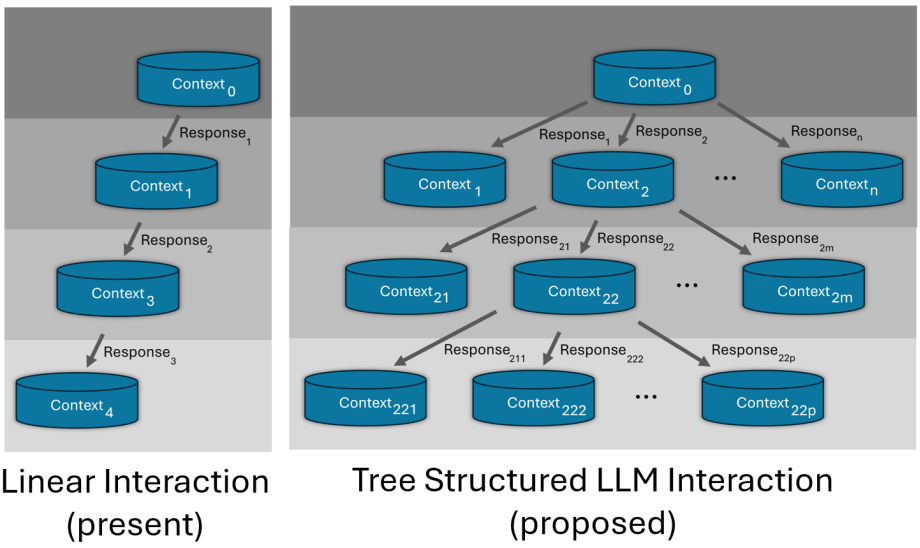

Fig. 4. Our current approach uses a linear interaction pattern which grows the LLM context with each LLM response until a successful answer is discovered or number of attempts is exhausted. One future direction is investigating how to run parallel LLM attempts organized in a tree.

Improving Contextual Framing Schema translation to familiar canonical domains (e.g., employees/products) may help the LLM reason about subtle semantic differences [6, 7].

Automatic Prompt Optimization Automating prompt search (e.g., via programmatic prompt optimizers) shows early promise for this task [1].

6 Conclusion

Faced with the reality that formal verifiers cannot handle many modern SQL features and that LLMs can be unreliable, we have developed a hybrid approach. Our system uses an LLM to robustly rewrite queries into verifiable forms and then employs a SMT based formal verifier to find counterexamples that prove non-equivalence.

The system has a number of limitations and we have shared several research directions to to make it more powerful, robust, and efficient. Despite its limitations experiments with a dataset of 23,994 LeetCode query pairs show that we can improve the coverage of a SOTA verifier and find new counterexamples that the verifier missed. Our approach extends the practical reach of formal methods, offering a promising path for computer aided verification of database systems.

References

1. Agrawal, L.A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M.J., Jiang, M., Potts, C., Sen, K., Dimakis, A.G., Stoica, I., Klein, D., Zaharia, M., Khattab, O.: GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning (2025), https://arxiv.org/abs/2507.19457
2. Chu, S., Wang, C., Weitz, K., Cheung, A.: Cosette: An automated prover for SQL. In: 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org (2017), http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf
3. Chu, S., Weitz, K., Cheung, A., Suciu, D.: HoTTSQL: proving query rewrites with univalent SQL semantics. SIGPLAN Not. **52**(6), 510–524 (Jun 2017). https://doi.org/10.1145/3140587.3062348, https://doi.org/10.1145/3140587.3062348
4. Dharwada, S., Devrani, H., Haritsa, J., Doraiswamy, H.: Query Rewriting via LLMs (2025), https://arxiv.org/abs/2502.12918
5. He, Y., Zhao, P., Wang, X., Wang, Y.: VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. Proc. ACM Program. Lang. **8**(OOPSLA1) (Apr 2024). https://doi.org/10.1145/3649849, https://doi.org/10.1145/3649849
6. Hua, W., Zhu, K., Li, L., Fan, L., Lin, S., Jin, M., Xue, H., Li, Z., Wang, J., Zhang, Y.: Disentangling logic: The role of context in large language model reasoning capabilities (2024), https://arxiv.org/abs/2406.02787
7. Lampinen, A.K., Dasgupta, I., Chan, S.C.Y., Sheahan, H.R., Creswell, A., Kumaran, D., McClelland, J.L., Hill, F.: Language models, like humans, show content effects on reasoning tasks. PNAS Nexus **3**(7), pgae233 (jul 2024). https://doi.org/10.1093/pnasnexus/pgae233
8. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proceedings of the VLDB Endowment **9**(3), 204–215 (2015)
9. Li, Z., Yuan, H., Wang, H., Cong, G., Bing, L.: LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. PVLDB **18**(1), 53–65 (2024). https://doi.org/10.14778/3696435.3696440
10. Miao, Z., Roy, S., Yang, J.: Explaining wrong queries using small examples. In: Proceedings of the 2019 International Conference on Management of Data. pp. 503–520 (2019)
11. OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., et al, J.A.: Gpt-4 technical report (2024), https://arxiv.org/abs/2303.08774
12. Singh, R., Bedathur, S.: Exploring the Use of LLMs for SQL Equivalence Checking (2024), https://arxiv.org/abs/2412.05561

13. Sun, Z., Zhou, X., Li, G.: R-Bot: An LLM-based Query Rewrite System (2024), https://arxiv.org/abs/2412.01661
14. Tan, J., Zhao, K., Li, R., Yu, J.X., Piao, C., Cheng, H., Meng, H., Zhao, D., Rong, Y.: Can large language models be query optimizer for relational databases? (2025), https://arxiv.org/abs/2502.05562
15. Veanes, M., Tillmann, N., de Halleux, J.: Qex: Symbolic SQL Query Explorer. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 425–446. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
16. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., Narasimhan, K.: Tree of thoughts: Deliberate problem solving with large language models. Advances in neural information processing systems **36**, 11809–11822 (2023)
17. Yao, Z., Li, H., Zhang, J., Li, C., Chen, H.: A query optimization method utilizing large language models (2025), https://arxiv.org/abs/2503.06902
18. Zhao, F., Chen, J., Lim, L., Ahmad, I., Agrawal, D., Abbadi, A.E.: LLM-SQL-Solver: Can LLMs Determine SQL Equivalence? (2025), https://arxiv.org/abs/2312.10321
19. Zhou, Q., Arulraj, J., Navathe, S.B., Harris, W., Wu, J.: SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE). pp. 2735–2748 (2022). https://doi.org/10.1109/ICDE53745.2022.00250