

SqlRewriter: Harnessing Community Knowledge to Rewrite SQL Queries

Qiushi Bai, Yihong Yu, Colin Michael Harrison, Jiatong Liu, James Liu, Jessie He, Hartley Tran, Jun Xia, Chen Li

UC Irvine, California 92697, USA

{qbai1, yihony1, colinmh2, jiatonl9, jyunrl, jessieh9, hartlemt, xiaj8}@uci.edu, chenli@ics.uci.edu

Abstract—SQL rewriting in databases is crucial for optimizing query performance by transforming a query to another that is equivalent but can be executed more efficiently. Yet, rewrite knowledge remains scattered in different domains and applications, limiting users’ access to valuable information to improve their queries. In this demo, we present SqlRewriter, a platform to support an open community for users to contribute and share SQL rewrite rules. We present novel solutions to challenges in developing the platform, including rule-quality control, user engagement and contribution, and system reliability. We use example cases to demonstrate the rich features and capabilities of the platform.

Index Terms—Databases, SQL, Query-Rewrite Rules, Communities

I. INTRODUCTION

Performance in database systems is crucial for applications that require fast responses to make mission-critical decisions. SQL query rewrite [1], [2] is an optimization technique that transforms a query to another that can produce the same results but have higher efficiency. Despite extensive research on query rewrite as a component of a database’s optimizer [1], [3], recent studies [4], [5] have indicated that solely depending on traditional database rewrites is inadequate for optimizing modern query workloads. As an example, the widespread adoption of business intelligence systems (e.g., Tableau and PowerBI) and Web applications built with object-relational mapping (ORM) frameworks lead to machine-generated queries. These queries are often challenging for databases to optimize [4]. Consequently, domain-specific knowledge and human-crafted rewriting rules are required to optimize workloads from these applications [6]. Query-rewrite knowledge is currently highly fragmented; it is scattered across informal venues such as technical blogs and forums (e.g., Stack Overflow), and it remains proprietary and inaccessible within commercial database optimizers (e.g., Calcite [7] and PostgreSQL). The lack of a centralized platform hinders database practitioners from sharing the rules and collaboratively innovating new rewrite techniques. While large language models (LLMs) are explored to enhance the query-rewrite process, their effectiveness still fundamentally relies on the quality and volume of existing rewrite knowledge prepared by human experts [8], [9]. To fully realize the potential of AI models in this domain, the discovery of novel rewrite knowledge becomes even more important than before.

In this paper, we demonstrate a system, SqlRewriter¹, to address these critical needs by leveraging the “wisdom of the crowd.” In particular, SqlRewriter harnesses the power of a community to collaboratively share SQL query-rewrite knowledge and innovate rules. It is designed to provide database practitioners with an easy-to-use Web-based platform for sharing existing knowledge and jointly discovering new rewrite rules. To the best of our knowledge, SqlRewriter is the first platform exclusively designed for sharing and innovating SQL query-rewrite knowledge.

Developing SqlRewriter required overcoming significant challenges to ensure its effectiveness, quality, and sustainability. The first challenge, *C1: Rule-Quality Control*, is ensuring the quality and correctness of community-contributed rules. We solve this challenge by adding quality-control “gates” throughout the lifecycle of a rule. In particular, SqlRewriter automatically suggests a rewrite rule based on an example query pair to minimize human errors. Each rule requires an approval process before becoming public to the community. Furthermore, after a rule is published, a community rating mechanism (e.g., thumbs-up/thumbs-down) allows users to assess the quality of the rule, and we allow rules to be revoked if issues arise. The second challenge, *C2: User Engagement and Contribution*, is encouraging active participation and contribution by community users. To address this challenge, we develop an intuitive interface for users to optimize their queries directly using community rules. We also implement public-facing user profiles that transparently display each user’s contributions, submitted rules, usage statistics, ratings, and comments. Finally, *C3: System Reliability*, focuses on a reliable service for many concurrent users. We achieve this goal using a Kubernetes-based architecture for scalability, a thorough CI/CD pipeline for stable feature deployment, a React-based frontend, and an agile Unicorn-based Python backend.

Related Work: Commercial systems such as EverSQL [10], SQLFlash [11], and Keebo [12] use AI models to suggest rewritten queries to user queries, but their rewrite knowledge is proprietary. Research studies such as Bao [13], Galo [14] and LearnedRewrite [15] leverage ML techniques to rewrite queries for better performance and efficiency. WeTune [4] generates new rewriting rules automatically by searching the

¹<https://sqlrewriter.io>

logical-plan space and considering the performance of rewritten queries. GRewriter [16] discovers new rewrite rules and selects equivalent rules atop the optimizer per-query basis for cloud-database customers. Postgres and MySQL provide a plugin for users to define rules to rewrite queries inside their optimizer. Users of these systems can benefit from the rewrite knowledge collected in SqlRewriter. This system uses a rule-definition language in our prior research [5], [6].

II. SYSTEM OVERVIEW

A. Architecture

Fig. 1 illustrates the architecture of SqlRewriter, deployed in a Kubernetes cluster for scalability and reliability. Users interact with the system through multiple flows: rewriting queries, contributing or publicizing rules via the approval workflow, discussing rewrite knowledge through comments, rating rules or rewrites via feedback, and viewing their community impact through user profiles. Incoming requests are load-balanced by the Ingress Controller, while the backend hosts the core application logic, including modules such as the Query Rewriter, Rule Generator, Rule Manager, User Manager, Workflow Manager, and Comment Manager. The Query Rewriter applies public rules from the Rule Base to optimize a user-provided query, and the Rule Generator suggests new, high-quality rewrites from user-provided examples. User accounts and authentication are managed by the User Manager and stored in the User Base. The modular architecture, combined with the Kubernetes deployment, enables flexible routing, persistent data storage, and concurrent, high-availability operations across all the services.

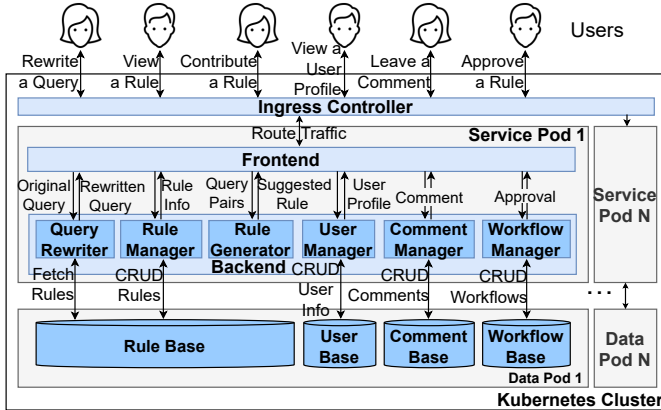


Fig. 1: System architecture of SqlRewriter.

B. Application Logic

Query Rewriter automatically rewrites a user input SQL query based on available rewrite rules (Figure 2). It parses the query into an Abstract Syntax Tree (AST) and fetches rewrite rules visible to the user from Rule Base as candidates. It parses the candidate rewrite rules into ASTs and then leverages the rewriting engine [5] to iteratively apply rules to the query. For each rule, it compares the rule's pattern against the query. If they match, it rewrites the current query into a rewritten

query, which becomes the current query for the next iteration. The iteration stops when no rules match the current query. It memorizes the rewriting sequence to break cycles (a rewritten query has been seen in the sequence) and stores the sequence to the database.

Rule Generator takes user-provided pairs of SQL queries as rewriting examples, explores different ways of generalizing tables, columns, expressions, predicates and subqueries into variables and suggests the most general rewrite rule to the user. The user may modify the rule using the rule-definition language introduced in [5] and submit the rule to the Workflow Manager to request approval before the rule becomes publicly available.

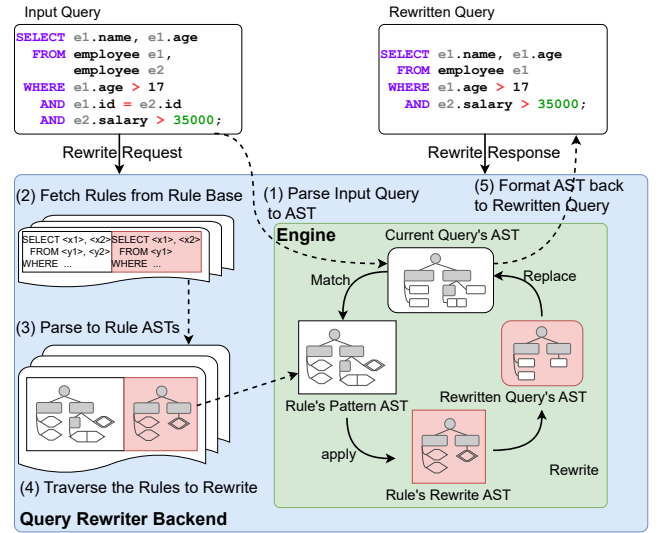


Fig. 2: Query rewriter application logic.

III. DEMONSTRATION SCENARIOS

In this section, we demonstrate two scenarios showing how users interact with SqlRewriter to create and contribute rules and benefit other users.

A. Creating and Sharing Rewrite Rules

This scenario illustrates how two users can mutually benefit from community-contributed rewrite rules. Suppose Alice is a Database Administrator (DBA) managing her company's sales dashboard application and enjoys sharing SQL rewrite knowledge with the community. One day, she discovers a rewrite that significantly accelerates a sales report query by transforming a subquery into a join query. She visits SqlRewriter and creates a rule by providing an example query pair. The system automatically suggests a rewrite rule in VarSQL [5], a simple yet expressive rule language that satisfies the example rewrite (Fig. 3). After verifying its correctness, she submits the rule (*Rule-A: Subquery to Join*) and requests to make it public. Once approved by the administrator, the rule is added to the community knowledge base.

Under the hood, each new rule is initially *private* and visible only to its creator. To make a rule publicly accessible and

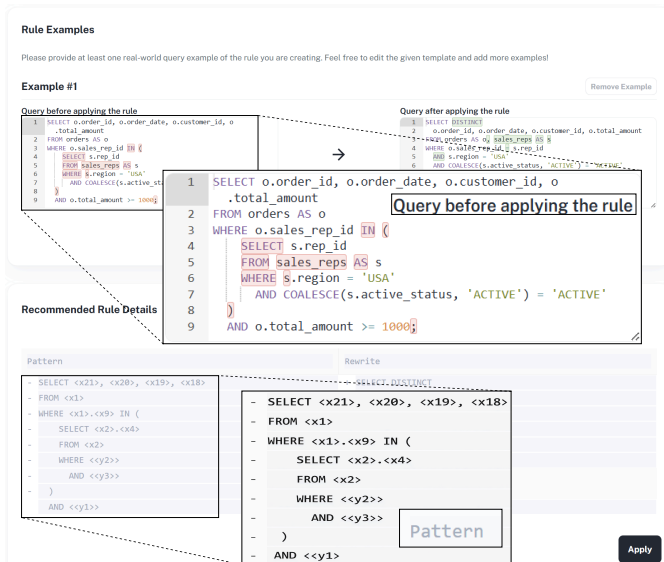


Fig. 3: Screenshot of SqlRewriter automatically suggesting a rewrite rule based on Alice's example.

applicable to other users' queries, the creator must complete the approval workflow. A rule remains in a *pending* state until it is approved by administrators and becomes *public*. It reverts to private if the creator modifies it or if administrators revoke it. Administrators can also reject low-quality rules and provide feedback through comments.

Now, Bob, a developer in the IT department at a university, wants to see how SqlRewriter can help him optimize a query from his Human Resource application. After he pastes an interesting query Q_2 into the "Rewrite Your Query" page, the system suggests a rewritten query Q_2' by applying Alice's *Rule-A: Subquery to Join* (Fig. 4). Bob finds that the rewritten query Q_2' is indeed more efficient than the original Q_2 and notices an additional opportunity to optimize it further by replacing the `DISTINCT` operator with a `GROUP BY` clause. Encouraged by SqlRewriter and Alice's rule, he decides to contribute his own rewrite to the community. Using the "Create a Rule" page and the system's rule generator, Bob submits his rule *Rule-B: Distinct to Group-By* to SqlRewriter.

After Bob's *Rule-B* becomes public, Alice checks whether any new community contributions can further optimize her original query. She pastes Q_1 into the **Rewrite Your Query** page, and the system suggests a new rewritten query Q_1'' that applies both *Rule-A* and *Rule-B* (Fig. 5). Impressed by Bob's optimization, Alice gives a thumbs-up rating to his *Rule-B*.

As shown in this scenario, SqlRewriter empowers users such as Alice and Bob to share and benefit from community rewrite knowledge. The system also ensures rule quality through automatic suggestions during rule creation and an approval workflow for publicizing rules.

B. Collaboration on Rewrite Rules

This scenario illustrates how two users collaborate through community discussions to refine and contribute a high-quality

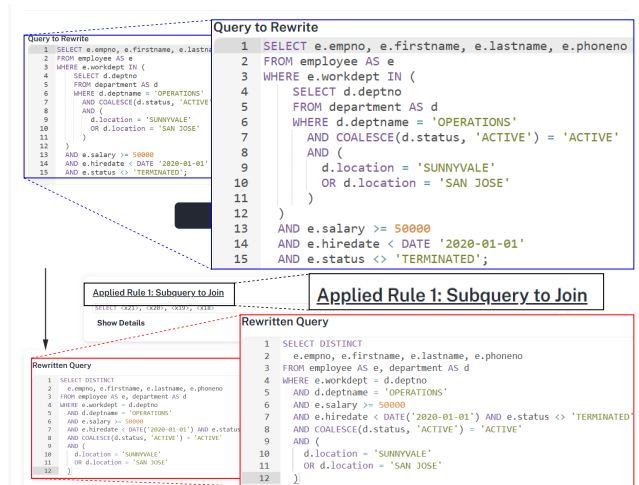


Fig. 4: Screenshot of SqlRewriter automatically suggesting a rewritten query Q_2' for Bob's query Q_2 using Alice's *Rule-A*.

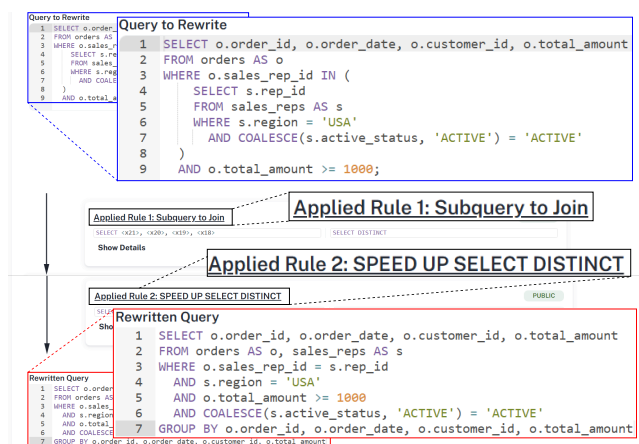


Fig. 5: Screenshot of SqlRewriter automatically suggesting a rewritten query Q_1'' for Alice's original query Q_1 , applying both her rule and Bob's contribution.

rule, showing how SqlRewriter supports the innovation and evolution of rewriting knowledge. Suppose Alice encounters a complex query while optimizing her company's gaming analytics application. She notices that queries using `SELECT DISTINCT ON` with `COALESCE` and multiple `EXISTS` subqueries often perform poorly. After testing, she rewrites these queries by removing the `EXISTS` filters, producing faster queries that return the same results. Based on this finding, Alice creates a new rule, *Rule-C: Optimize SELECT DISTINCT ON with COALESCE* (Fig. 6), and requests to make it public. Because the rule is syntactically correct and captures a nontrivial optimization, the administrator approves it, and *Rule-C* becomes public.

Later, Bob finds Alice's *Rule-C* to be interesting. Curious about its behavior, he applies it to one of his test cases and observes that, while the performance indeed improves, the rewritten query returns more rows than the original. Recognizing that the rule may have altered the query semantics by including rows that should be filtered out, Bob comments

```

SELECT DISTINCT
ON (<x1> <x2>, <x3>, <x1>, COALESCE((
  SELECT <x4>.<x5>
  FROM <x4>
  WHERE <<x9>>
  AND <<x21>>
  LIMIT 1
), <x6> )
FROM <x8>
WHERE <x23>
AND <x24>
AND EXISTS (
  SELECT 1
  FROM <x4>
  WHERE <<x9>>
  AND <<x21>> )
ORDER BY <x8>.<x22> DESC;

```

(a) Alice's *Rule-C*'s pattern.

```

SELECT DISTINCT
ON (t.playerId) t.gzipId, t.pubCode, t.
  playerId, COALESCE((
    SELECT p.preferenceValue
    FROM userPreferences AS p
    WHERE t.gzipId = p.gzipId
    AND p.preferenceValue IN ('en', '
    hi')
  ), 'en')
FROM userPlayerIdMap AS t
WHERE t.pubCode IN ('hymas', 'ayqioa', '
  rj49as99')
AND t.provider IN ('FCM', 'ONE_SIGNAL')
AND EXISTS (
  SELECT 1
  FROM userPreferences AS p
  WHERE t.gzipId = p.gzipId
  AND p.preferenceValue IN ('en', 'hi
  hi')
)
ORDER BY t.playerId DESC;

```

(c) Alice's *Rule-C*'s example original query.

Fig. 6: Alice's initial contribution of *Rule-C*: *Optimize SELECT DISTINCT ON with COALESCE*, showing the pattern–rewrite template on top and her example original–rewritten query pair on the bottom.

directly on *Rule-C*'s discussion page to highlight the issue. Notified by the system, Alice reviews Bob's feedback, identifies the problem, and responds with a potential fix (Fig. 7).

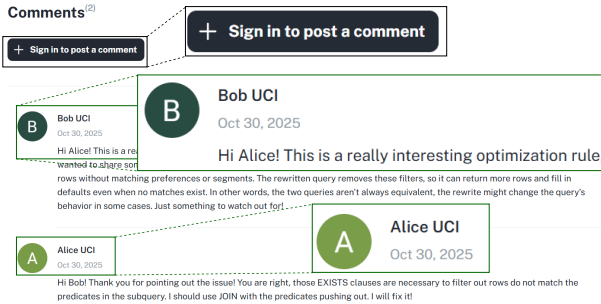


Fig. 7: Screenshot showing Bob's comment on potential semantic issues in Alice's *Rule-C*: *Optimize SELECT DISTINCT ON with COALESCE* and Alice's reply with suggestions to improve the rewrite.

After revising the rule, Alice updates *Rule-C* to preserve the filtering semantics by introducing *LEFT JOIN* while keeping the *DISTINCT ON* structure. Because edits to a public rule automatically revert it to private, she resubmits it for approval. After it is approved again, Bob tests the updated rule and finds that *Rule-C* now correctly preserves the semantics while maintaining performance benefits. Satisfied with the improvement, he leaves a comment praising Alice's refinement and gives the rule a thumbs-up. Alice views her contributions on her "Profile" page and notices Bob's thumbs-up on *Rule-C*. The profile also highlights her active participation in discussions

```

SELECT DISTINCT
ON (<x1>) <x2>, <x3>, <x1>, COALESCE(
  (
    SELECT <x4>.<x5>, <x6>
    FROM <x4>
    WHERE <<x9>>
    AND <<x21>>
    LIMIT 1
  ), <x6> )
FROM <x8>
WHERE <x23>
AND <x24>
AND <<x21>>
ORDER BY <x8>.<x22> DESC;

```

(b) Alice's *Rule-C*'s rewrite.

```

SELECT DISTINCT
ON (t.playerId) t.gzipId, t.pubCode,
  t.playerId, COALESCE(p.
    preferenceValue, 'en'), s.
    segmentId
FROM userPlayerIdMap t
LEFT JOIN userPreferences p ON t.
  gzipId = p.gzipId
WHERE t.pubCode IN ('hymas', 'ayqioa
  ', 'rj49as99')
AND t.provider IN ('FCM', '
  ONE_SIGNAL')
AND p.preferenceValue IN ('en', 'hi
  hi')
ORDER BY t.playerId DESC;

```

(d) Alice's *Rule-C*'s example rewritten query.

and her accumulative contributions to the public rule base.

This scenario demonstrates how *SqlRewriter* encourages collaborative innovation of query rewriting knowledge. Users like Alice and Bob can discuss and refine rewrite rules to ensure both correctness and performance improvement, supported by community feedback mechanisms. This workflow not only ensures high-quality rules through re-approval and transparent discussions but also motivates users to contribute more by recognizing their community impact through profile-based contribution tracking.

ACKNOWLEDGMENTS

We thank Justin Huang, Spencer Peng, and Seongjin Yoon for their contributions to the system.

REFERENCES

- [1] B. F. and G. G., "A rule-based query rewriter in an extensible DBMS," in *ICDE 1991*, 1991, pp. 248–256. [Online]. Available: <https://doi.org/10.1109/ICDE.1991.131472>
- [2] H. P., J. H., and W. H., "Extensible/rule based query rewrite optimization in starburst," in *SIGMOD 1992*. ACM Press, 1992, pp. 39–48. [Online]. Available: <https://doi.org/10.1145/130283.130294>
- [3] G. Graefe, "Volcano - an extensible and parallel query evaluation system," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 1, pp. 120–135, 1994.
- [4] Z. Wang, Z. Zhou, and et al., "Wetune: Automatic discovery and verification of query rewrite rules," in *Proc. of SIGMOD 2022*. ACM, 2022, pp. 94–107. [Online]. Available: <https://doi.org/10.1145/3514221.3526125>
- [5] Q. Bai, S. Alsudais, and C. Li, "Querybooster: Improving sql performance using middleware services for human-centered query rewriting," *Proc. VLDB Endow.*, vol. 16, no. 11, p. 2911–2924, Jul. 2023. [Online]. Available: <https://doi.org/10.14778/3611479.3611497>
- [6] —, "Demo of querybooster: Supporting middleware-based sql query rewriting as a service," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 4038–4041, Aug. 2023. [Online]. Available: <https://doi.org/10.14778/3611540.3611615>
- [7] Apache Calcite, <https://calcite.apache.org/>.
- [8] Z. Li, H. Yuan, H. Wang, G. Cong, and L. Bing, "Llm-r2: A large language model enhanced rule-based rewrite system for boosting query efficiency," *Proc. VLDB Endow.*, vol. 18, no. 1, p. 53–65, Sep. 2024. [Online]. Available: <https://doi.org/10.14778/3696435.3696440>
- [9] Z. Sun, X. Zhou, G. Li, X. Yu, J. Feng, and Y. Zhang, "R-bot: An llm-based query rewrite system," *Proc. VLDB Endow.*, vol. 18, no. 12, p. 5031–5044, Sep. 2025. [Online]. Available: <https://doi.org/10.14778/3750601.3750625>
- [10] <https://www.eversql.com/>, last accessed: 2023-01-07.
- [11] <https://sqlflash.ai/>, last accessed: 2025-10-27.
- [12] <https://keebo.ai/>, last accessed: 2023-01-07.
- [13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 1275–1288. [Online]. Available: <https://doi.org/10.1145/3448016.3452838>
- [14] G. Damasio, V. Corvinelli, P. Godfrey, P. Mierzejewski, A. Mihaylov, J. Szlichta, and C. Zuzarte, "Guided automated learning for query workload re-optimization," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2010–2021, 2019. [Online]. Available: <http://www.vldb.org/pvldb/vol12/p2010-damasio.pdf>
- [15] X. Zhou, G. Li, C. Chai, and J. Feng, "A learned query rewrite system using monte carlo tree search," *Proc. VLDB 2021*, vol. 15, no. 1, pp. 46–58, 2021. [Online]. Available: <http://www.vldb.org/pvldb/vol15/p46-li.pdf>
- [16] Z. Jiang, Z. Wang, H. Lan, C. Tang, H. Ding, L. Wang, S. Zou, Z. Wei, Y. Liu, X. Yu, Y. Ren, G. Li, and H. Chen, "Grewriter: Practical query rewriting with automatic rule set expansion in gaussdb," *Proc. VLDB Endow.*, vol. 18, no. 12, p. 4991–5003, Sep. 2025. [Online]. Available: <https://doi.org/10.14778/3750601.3750622>