Query running too slow? Rewrite it with Quorion!

Bingnan Chen
bchenba@ust.hk
Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Binyang Dai
bdaiab@ust.hk
Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Qichen Wang
qichen.wang@epfl.ch
EPFL
Lausanne, Switzerland

Ke Yi
yike@ust.hk
Hong Kong University of Science and Technology
Hong Kong, Hong Kong

ABSTRACT

We will demonstrate Quorion, a query rewriter with theoretical guarantees and better practical performance. Quorion adopts some of the recently developed query planning methods that provide optimality guarantees, including Yannakakis$^+$, an optimized version of the Yannakakis algorithm, generalized hypertree decompositions (GHD), GYO reduction, and cost-based optimization. Quorion also provides a platform for users to explore different query plans for a given query through a web-based interface and compare their performance with classical query plans. Quorion currently supports DuckDB, MySQL, and PostgreSQL, and can be connected to any user-provided database easily through a JDBC connector.

KEYWORDS

conjunctive query, query rewrite, compilation, visualization

PVLDB Reference Format:
Bingnan Chen, Binyang Dai, Qichen Wang, and Ke Yi. Query running too slow? Rewrite it with Quorion! . PVLDB, 18(12): 5243 - 5246, 2025.
doi:10.14778/3750601.3750642

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hkustDB/Quorion.

1 INTRODUCTION

Queries involving selection, join, projection, and aggregation across multiple tables, namely SPJA queries, play a significant role in Analytical Processing (AP). For instance, a user may want to query the total amount (aggregation) spent and the spending in different applications (projection) using the same identity across various software (join) within a specific time period (selection). Efficiently executing such SPJA queries is one of the fundamental issues in database research. However, in many cases, the performance of these queries is often unsatisfactory, especially when dealing with large volumes of data and/or when the query involves complicated joins. To address these challenges, many new query planning algorithms have been proposed with good theoretical guarantees, but none of them has made their way to production database systems due to the difficulty of implementation and, for some algorithms, many radical changes to the query engine.

To leverage on the new query planning algorithms without changing the codebase of existing SQL engines, we have developed Quorion, a query writer that implements the new algorithms by rewriting a given query into multiple SQL statements, which can then be passed to any SQL engine for execution.

Example 1.1. Consider a graph represented by a table with two attributes: src and dst. The following query counts the number of length-4 paths in this graph:

```sql
SELECT COUNT(*)
FROM Graph AS g1, Graph AS g2, Graph AS g3, Graph AS g4
WHERE g1.dst = g2.src AND g2.dst = g3.src AND g3.dst =
     g4.src
```

For this query, all existing DBMSs would first compute the join, and then count them. This query plan has a worst-case cost of $O(N^3)$, where N is the number of tuples in the relation Graph. On the other hand, it is known that this query can be actually evaluated in $O(N)$ time using the Yannakakis algorithm [4]. On this query, the query plan generated by Quorion precisely implements (an improved version of) this algorithm [3], in the form of 7 SQL statements (please see Section 4 for details). The theoretical advantage also translates into practice: When feeding this query (unmodified) to DuckDB on a graph with 73, 233 edges, it takes DuckDB 97.234 seconds to finish. After the rewriting, the time drastically reduces to 0.021 seconds. Similar improvements are also observed on other DBMSs such as MySQL and PostgreSQL.

In addition to Yannakakis$^+$ [3], the improved version of the Yannakakis algorithm, Quorion has also implemented generalized hypertree decompositions (GHD) [2], the GYO reduction algorithm [5], as well as a cost-based optimizer. This allows Quorion to handle a large class of SPJA queries, covering most of the commonly used AP benchmarks, with noticeable improvements for a majority of them.

Meanwhile, we also make Quorion easy to use: After users submit queries to the system through a visual interface, the system will first adopt these state-of-the-art methods to generate the improved query plans. Quorion then visualizes these query plans with additional information, such as the estimated cost for better plan selection. The system will then generate a set of rewritten SQL

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750642

statements following the designated query plan and execute inside any given SQL engine.

We hope to achieve the following goals through this demonstration:

- At the very least, Quorion provides another option for slow queries. When users encounter such a case, they can try Quorion with ease. Quorion is designed to be extensible. It currently supports DuckDB, PostgreSQL, and MySQL, but can be extended to any SQL engine using a JDBC connector.
- Our demonstration allows users to visualize and choose between different query plans, with cost statistics derived from our optimizer. This serves pedagogy purposes, facilitating users in understanding the optimization process and recognizing the advantages of the new query plans.
- While these new query algorithms, especially those with good theoretical guarantees, have been known for decades and tested using custom implementations, they have seen little adoption in practice. As Quorion provides an easy way for people to try them out in the various DBMSs, we hope this would improve the awareness and pave the way to their eventual adoption.

2 YANNAKAKIS$^+$ QUERY PLANNER

The traditional query plan is a DAG, where each leaf node represents an input relation, and internal nodes represent relational algebra operations. The edges in the DAG denote the inputs and outputs of these relational algebra operations. Yannakakis$^+$ is also pure relational, in the sense that it can be formulated as a DAG query plan consisting of standard relational operators. The technical details for Yannakakis$^+$ are available in [3]. Below are some high-level insights regarding how Yannakakis$^+$ transforms from the native query into a series of equivalent rewritten queries.

Core Mechanism. Yannakakis involves two rounds of semi-join and one round of joins, while Yannakakis$^+$ planner can further improve the query plan to a round of bottom-up semi-joins, followed by a round of top-down join. Given a join tree $\mathcal{T}$ with a designated root node, Yannakakis$^+$ first computes the semi-joins or join-aggregates between every node R and its parent node R_p using a post-order traversal of $\mathcal{T}$. Once this first round is complete, it then performs join aggregation between each node R and all of its child nodes R_c, following a pre-order traversal of $\mathcal{T}$.

Compared to the Yannakakis algorithm, Yannakakis$^+$ reduces the number of semi-join and join operations. In the optimal scenario, when the query is relation-dominated [3], Yannakakis$^+$ only requires one round of join-aggregates, which significantly decreases the overhead associated with the Yannakakis algorithm.

Aggregation Query Strategy. Moreover, for aggregation queries, we push some join-aggregation operations down before the semi-joins as much as possible. This is crucial because aggregations can significantly reduce the data size, especially for queries with a small output size M, while each join can eliminate a relation. An aggregation query example is shown in Example 4.1.

Query Plan Optimization. Since the rooted join tree may not be unique for a given query, our rewriter produces a set of query plans instead of just one. All of these plans have the same asymptotic running time. Therefore, it is crucial to choose an optimal (or near-optimal) plan from this set using rules and data statistics. To achieve this, we have designed a query optimizer specifically tailored to our algorithm. We estimate the cost of various execution plans by considering factors such as cardinalities, the number of distinct values (NDV), and quantiles. Additionally, we have incorporated effective rule-based optimizations. For example, semi-join and aggregation elimination help remove redundant operations in cases involving primary key and foreign key (PK-FK) joins, which significantly improves rewrite efficiency.

3 SYSTEM ARCHITECTURE

We have implemented a system with four major components: web-based interface, parser & planner, optimizer (CBO & RBO), and rewriter. The system architecture is shown in Figure 1. The front-end interacts with the back-end, and the parser & planner communicates with the optimizer & rewriter via REST APIs. Below are the details of each component:

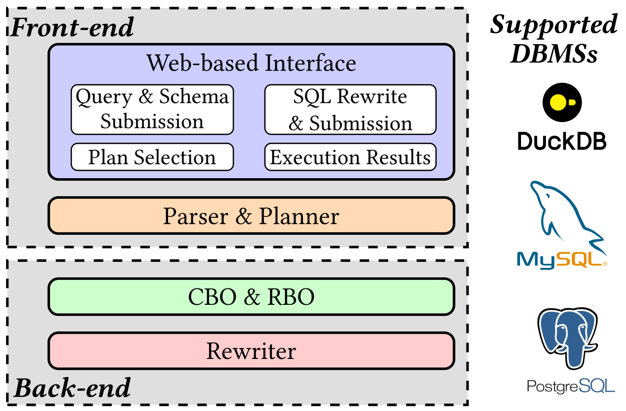


Figure 1: Quorion system architecture

Web-based interface. We have created a web-based user interface that offers an interactive and visual experience. This interface allows users to follow the phases required to use our system effectively with ease and clarity. It accepts user input, including queries and schema, for parsing and planning.

Parser & Query planner. In this component, we utilize the parser and planner from [1]. Our system can support a wide range of queries for parsing. With the exception of a small number of nested queries that are not currently supported (such as more complex ones where the nested part involves joins), our parser can process all other queries. The parser transforms SQL queries into logical plans, which are then converted into relational hypergraphs. If the query is cyclic, the GHD algorithm is employed; if it is acyclic, the GYO algorithm is used to generate candidate join trees. Once the join trees are created, additional query information, such as aggregations and projections, is added on top of them. The resulting plan is then sent to the optimizer for cost and rule-based optimization.

CBO & RBO. The optimizer assesses the cost of different join orders based on the structure of the join tree and various statistics, such as cardinality and the number of distinct values (NDV) at each node. These statistics are provided by the DBMS. For complex queries that require extensive optimization times, our system allows the option to skip the planning and optimization steps. Instead, it can use the join tree supplied by the DBMS for the next rewrite phase. This approach helps to balance the trade-off between optimization time and query execution time.

Rewriter. Upon receiving the join trees, our system uses the Yannakakis$^+$ algorithm (introduced in Section 2) to convert these trees into a series of equivalent intermediate representations (IRs) and then translate them into executable SQL queries. More details of the algorithms can be found in [3]. Currently, we use temporary views to store the intermediate results of our plan, a strategy that aligns with our algorithm while adding minimal overhead. Finally, the SQL queries, along with their associated costs, are sent back to the front-end for display.

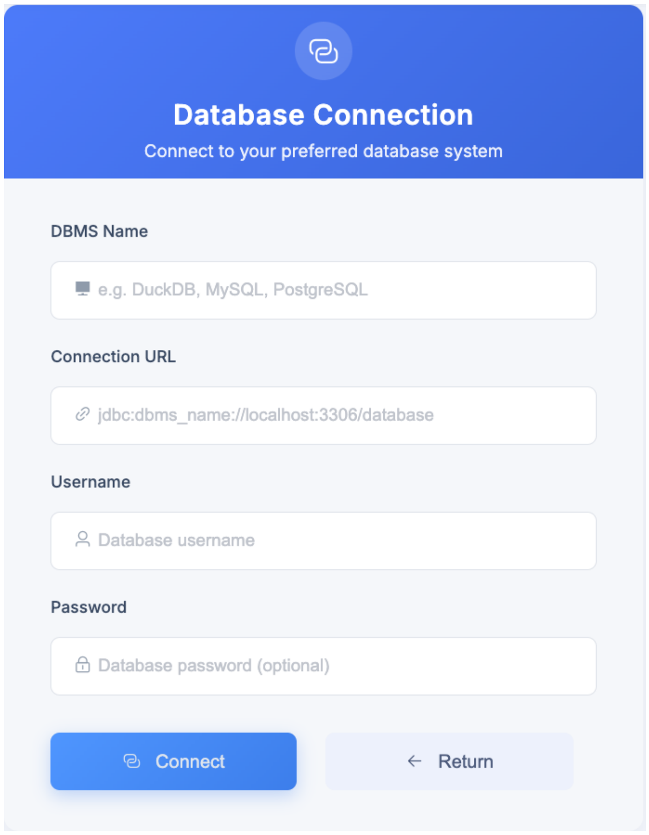

Database Connection
Connect to your preferred database system
DBMS Name
e.g. DuckDB, MySQL, PostgreSQL
Connection URL
jdbc:dbms_name://localhost:3306/database
Username
Database username
Password
Database password (optional)
Connect
Return

Figure 2: Add a new DBMS through JDBC.

DBMS. Users can select the optimal join tree based on the provided cost. The front-end will then display the generated rewritten queries, allowing users to choose different DBMSs to execute the queries. By default, we support three built-in types of DBMSs: DuckDB, MySQL, and PostgreSQL. To add a new DBMS, users simply need to set up the JDBC connector with the corresponding DBMS name, DBMS URL, username, and password, as shown in Figure 2. With the provided JDBC driver, users can submit queries to any other DBMSs that support SQL.

4 DEMONSTRATION

During the demonstration, we will offer an interactive experience that helps the audience gain a better understanding of next-generation query plans. Attendees will have the opportunity to experiment with various queries, datasets, and DBMSs to observe how these query plans outperform native ones. Specifically, we will guide the audience through the following steps:

Phase 1. Query & Schema Submission. Quorion is pre-built with Sub-Graph Pattern (SGPB) Benchmark, JOB Benchmark, TPC-H Benchmark, and LSQB Benchmark [3], users can use it directly, shown in Figure 3. If users want to use a custom schema or query, they can select the custom module, which allows for personalized input.

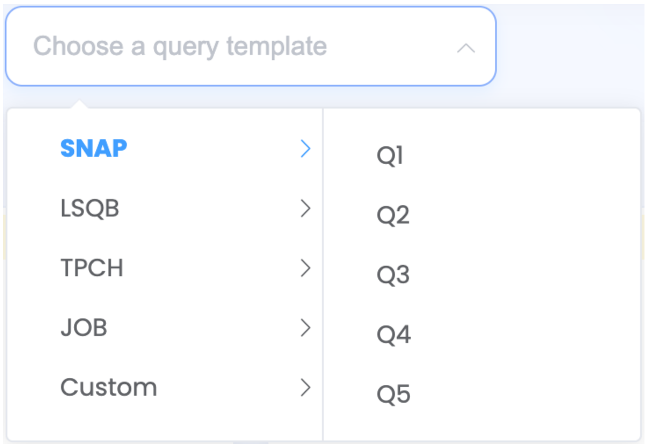
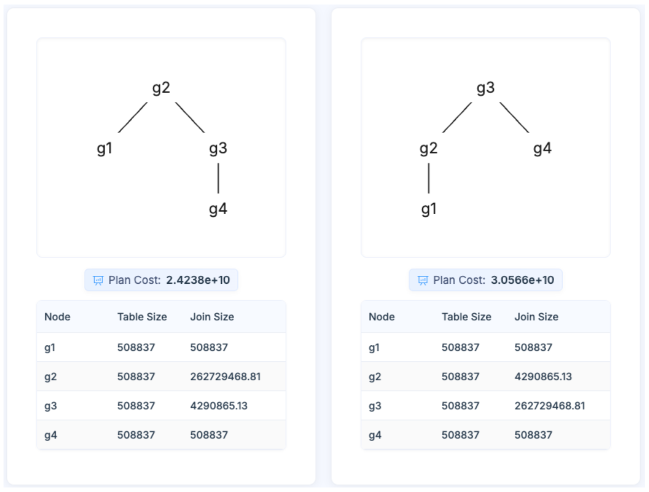


Figure 3: Query Selection

Phase 2. Plan Selection. After submitting queries and schemas, Quorion presents several candidate join trees, each accompanied by a series of statistical data to help users make appropriate choices. The given information includes: (1) Join tree: Represent the execution order of the rewrite operations. A bottom-up and top-down (optional) scan based on this tree will be conducted. (2) Plan Cost: Represent our estimated execution cost of the entire plan. A higher value indicates greater overhead, so it is preferable to choose a lower-cost join tree. (3) Detailed statistics for each node in the join tree: Table Size represents the cardinality of the relation corresponding to this node, and Join Size represents the final join size after the child nodes and this node have been joined. As shown in Figure 4, since the plan cost of the right join tree exceeds that of the left join tree (2.4238e+10 $<$ 3.0566e+10), the left join tree is chosen.

Phase 3. SQL Rewrite & Submission. With the selected join tree, Quorion performs the query rewrite and returns rewritten SQL queries. A more detailed step-by-step example is presented at Example 4.1. Next, users can submit and execute the native and rewritten queries to the designated DBMS.



Plan Cost: **2.4238e+10**

Node	Table Size	Join Size
g1	508837	508837
g2	508837	262729468.81
g3	508837	4290865.13
g4	508837	508837

Plan Cost: **3.0566e+10**

Node	Table Size	Join Size
g1	508837	508837
g2	508837	4290865.13
g3	508837	262729468.81
g4	508837	508837

Figure 4: Candidate Plan Selection.

Example 4.1. We elucidate the detailed process for generating the rewritten queries using Example 1.1. For this example, we opt for the join tree shown in Figure 5 to describe the procedure. This join tree has several valid reduction orders, and we selected one of them for the following steps.

Step 1. Reduce Node g4 to Node g3. We project g4 onto the `src` attribute and count the number of tuples as `annot`. The resulting temporary view is then joined with node g3, as shown in Figure 5.

```
CREATE OR REPLACE TEMP VIEW aggView1 AS SELECT src AS v6,
    COUNT(*) as annot FROM Graph AS g4 GROUP BY src;
CREATE OR REPLACE TEMP VIEW aggJoin1 AS SELECT src AS v4,
    annot FROM Graph AS g3, aggView1 WHERE
    g3.dst=aggView1.v6;
```

Step 2. Reduce Node g1 to Node g2. We project g1 onto the `dst` attribute and count the number of tuples, storing the result as `annot`. The temporary view created from this projection is then joined with node g2. This process is illustrated in Figure 5.

```
CREATE OR REPLACE TEMP VIEW aggView2 AS SELECT dst AS v2,
    COUNT(*) AS annot FROM Graph AS g1 GROUP BY dst;
CREATE OR REPLACE TEMP VIEW aggJoin2 AS SELECT dst AS v4,
    annot FROM Graph AS g2, aggView2 WHERE
    g2.src=aggView2.v2;
```

Step 3. Reduce Node g3 to Node g2. Following the same approach, we project and aggregate the query result from ***Step 1***, updating the annotation values. The result is then joined with g2.

```
CREATE OR REPLACE TEMP VIEW aggView3 AS SELECT v4,
    SUM(annot) AS annot FROM aggJoin1 GROUP BY v4;
CREATE OR REPLACE TEMP VIEW aggJoin3 AS SELECT
    aggJoin2.annot * aggView3.annot AS annot FROM
    aggJoin2 JOIN aggView3 USING(v4);
```

Step 4. Compute the Final Result. We aggregate all `annot` values to obtain the final result.

```
SELECT SUM(annot) FROM aggJoin3;
```

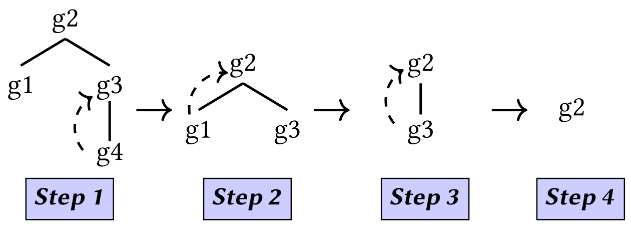


Figure 5: Rewrite steps for Example 1.1.

In summary, for aggregation queries, the rewritten queries break down every binary join into two steps. First, initial aggregation operations are conducted based on the join condition to gather important statistics. Second, the join is performed using the statistics obtained from the first step. By following these steps, we can push the aggregation function down as far as possible, which helps minimize the inflation of intermediate results.

Phase 4. Execution Results. Quorion will also gather the runtime for both the original query and the rewritten queries from the target DBMS. The results will be visualized in the web-interface in the form of a bar chart, as shown in Figure 6.

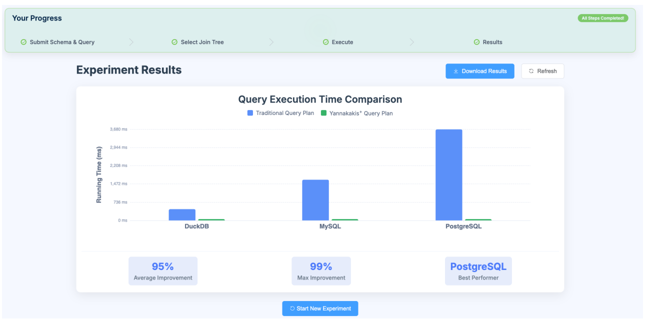


Figure 6: Execution results for SNAP-Q0.

REFERENCES

[1] Binyang Dai, Qichen Wang, and Ke Yi. 2023. SparkSQL+: Next-generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 115–118. https://doi.org/10.1145/3555041.3589715

[2] Georg Gottlob, Martin Grohe, nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree Decompositions: Structure, Algorithms, and Applications. In *Proceedings of the 31st International Conference on Graph-Theoretic Concepts in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1–15. https://doi.org/10.1007/11604686_1

[3] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 235 (may 2025), 28 pages. https://doi.org/10.1145/3725423

[4] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[5] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.* IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509