# QueryBrew: System-Agnostic SQL-to-SQL Query Optimization

Tobias Schmidt*
Technische Universität München
tobias.schmidt@in.tum.de

Maximilian Reif*
Technische Universität München
reif@in.tum.de

Altan Birler*
Technische Universität München
altan.birler@tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

## ABSTRACT

Efficient SQL execution heavily depends on the query optimizer's ability to find efficient query plans. Existing query optimizers differ widely in their capabilities, leading to significant differences in execution complexity across many queries. In this demonstration, we present QueryBrew, a system-agnostic query optimization service that decouples the optimizer from the database engine through SQL-to-SQL rewriting. QueryBrew takes arbitrary SQL queries, refines them using the state-of-the-art Umbra optimizer, and distills the optimized execution plan back into an operator-oriented SQL representation. This allows any target SQL database, such as PostgreSQL, DuckDB, ClickHouse, or SQL Server, to inherit advanced optimization techniques, such as general unnesting, simplification, and adaptive join ordering, without modification. The QueryBrew UI lets users explore and analyze the changes made by Umbra's query optimizer and how they affect execution in the target system. SQL-to-SQL optimization can improve runtimes by more than one order of magnitude for correlated and many-join queries.

**PVLDB Reference Format:**
Tobias Schmidt, Maximilian Reif, Altan Birler, and Thomas Neumann. QueryBrew: System-Agnostic SQL-to-SQL Query Optimization. PVLDB, 19(12): 4494 - 4497, 2026.
doi:10.14778/3827998.3828048

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://querybrew.db.cit.tum.de.

## 1 INTRODUCTION

Databases have gotten significantly faster over the last decades. Vectorized execution in DuckDB and SQL Server and compiled execution in PostgreSQL significantly speed up analytical queries. High-bandwidth storage devices and massively parallel machines allow processing multi-terabyte datasets on a single node. Distributed engines such as Snowflake, Databricks, BigQuery, and Redshift are widely deployed in the industry and support petabyte-scale analytics. However, one particular component has lagged behind this pace of innovation: the query optimizer.

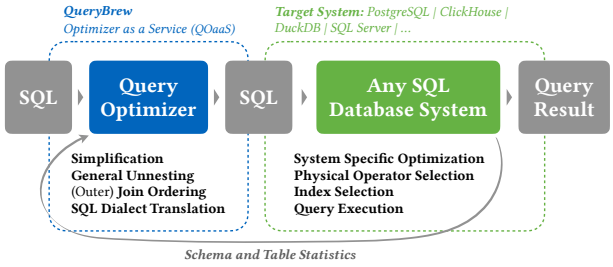


**Figure 1: QueryBrew's SQL-to-SQL optimization approach. Queries are optimized by a state-of-the-art optimizer (e.g., Umbra) and transformed back to SQL. The optimized query benefits from simplification rules, general unnesting, and advanced join ordering [5], and runs in existing SQL databases.**

The query optimizer is the component with the greatest potential to make or break queries. Its decisions not only affect runtime constants but can change a query's runtime complexity entirely. It is a high-risk and high-reward component. Changes to the optimizer can yield immense benefits, but they are also likely to break some customers' workloads in unexpected ways (e.g., by removing one of two mistakes that cancelled each other out). Thus, big players in the industry approach developments in the optimizer with a risk-averse, calculated approach. Unfortunately, this means that optimizers often lag behind the latest innovations [14].

Decoupling the optimizer from the database engine would accelerate innovation and enable independent development. However, optimizers are deeply intertwined with the database system, its statistics, and its operators; hence, extracting them is non-trivial. Even the many database solutions at Google, such as BigQuery, Spanner, F1, BigTable, Dremel, and Procella, share the SQL frontend GoogleSQL but do not share a single optimizer [2].

There have been attempts to design a common intermediate representation for query plans such as Substrait [1]. CompoDB [7] exploits Substrait to build a modular data system with an exchangeable query optimizer and execution engine. They rely on DuckDB, DataFusion, Calcite, and Ibis for optimization. However, only a few engines (DuckDB, DataFusion, and Acero) are supported, as Substrait has not seen wide adoption due to the inherent difficulty of unifying the semantics of existing systems with their various idiosyncrasies. Microsoft takes this idea one step further and proposes Query Optimizer as a Service (QOaaS). They make Fabric's Unified Query Optimizer [6] available to other engines, such as Spark, via Substrait-to-Substrait optimization. The logical plan is given to the optimizer, which produces an optimized, physical plan [15].

*Authors contributed equally to this research.


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828048

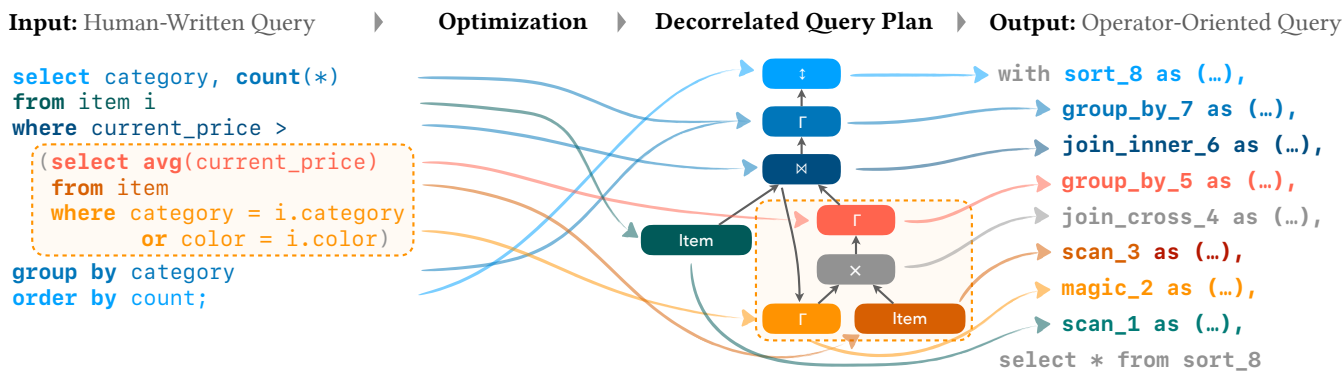


**Figure 2: QueryBrew in action: Correlated query on the TPC-DS `item` table. Umbra's query optimizer decorrelates the query using general unnesting. We transform the optimized query plan into an operator-oriented representation using one CTE per operator that can canonically be executed by other database systems.**

We approach the decoupling of query optimizers in a practical way as shown in Figure 1. Our optimization service, QueryBrew, takes SQL as input, optimizes the query internally, and brews SQL as output. We found that rewriting queries in SQL can greatly improve performance across a wide range of queries. Because most relational databases use SQL as their standard interface, QueryBrew integrates easily with existing systems. In addition, the target system's query optimizer can further apply system-specific optimizations and select the best physical operators, taking the system's implementation details into account.

We are not the first to explore SQL-to-SQL optimization. Both LLM-based [8] and human-centered [3] rewriting approaches have been proposed; however, these surface-level techniques operate on the query text or the abstract syntax tree. Since many sophisticated optimizations operate on relational algebra, the capabilities of such text-based techniques are inherently limited. OpenIVM [4] also operates on relational algebra, but it generates SQL for incremental view maintenance rather than for query optimization.

QueryBrew demonstrates that complex, correlated queries can be optimized directly within SQL. By leveraging schema and statistics, it simplifies, unnests, and utilizes cost-based reordering via adaptive optimizations and enumeration of join plans to find the best execution plan. Through a structured back-translation of the optimized query plan to SQL, target systems can benefit from these optimizations without reimplementing them. Additionally, an intuitive interface lets users easily analyze SQL-to-SQL optimizations and benchmark queries across different systems.

## 2 SOLUTION OVERVIEW

**Approach.** QueryBrew builds a refined SQL statement by passing an input query through Umbra's [11] state-of-the-art optimizer and distilling the resulting optimized plan back into SQL. Umbra implements a wide range of (cost-based) optimization techniques, including operator simplification, general unnesting [10, 12], adaptive join reordering [13], and common subtree elimination. These techniques exploit detailed statistics, data samples, type information, and functional dependencies available to the database to estimate cardinalities and simplify or eliminate operators. The resulting query plan is not a typical query tree but a directed acyclic graph (DAG) of relational operators, enabling more efficient execution.

QueryBrew translates the optimized query plan back to SQL, retaining the optimizations made by Umbra. It preserves the execution order of the operators from the optimized plan, allowing other systems to benefit from Umbra's join optimizer. We achieve this using Common Table Expressions (CTEs) in the exported SQL. Every operator in the optimized plan is represented as an individual CTE, and operators are connected by referencing the CTEs of their inputs. Consider the following query:

```
select i_category, count(distinct i_color)
from item group by i_category;
```

It counts the number of distinct colors for each category in the item table from the TPC-DS benchmark.

Umbra's optimized query plan translated to SQL looks as follows:

```
WITH scan_1 AS (
  SELECT i_category AS v5, i_color AS v6 FROM item
), groupby_2 AS (
  SELECT v5 AS v3, v6 AS v4 FROM scan_1 s GROUP BY v5, v6
), groupby_3 AS (
  SELECT v3 AS v1, count(v4) AS v2
  FROM groupby_2 s GROUP BY v3)
SELECT v1 AS i_category, v2 AS count FROM groupby_3;
```

Every operator from the plan is given a unique name (e.g., `scan_1`, `groupby_2`, `groupby_3`). While the scan reads the two required columns from the `item` table, the two group by operators compute the distinct count. Here we can already observe one of the optimizations performed by Umbra: It splits the distinct count into two group by operators (`groupby_2` and `groupby_3`), one to compute all distinct combinations of category and color, and another one to count the number of colors for each category. Thanks to Umbra's general unnesting, converting an operator is independent of the overall query structure: each operator simply references the CTEs of its inputs. The translation to SQL is canonical for most operators; only a few non-standard operators (group-joins, mark-joins, and magic sets) require special care.

**Integration.** The CTEs provide a structured, more optimized representation of the original queries, allowing the target systems to exploit some of Umbra's advanced optimization techniques without implementing them themselves. For instance, Umbra implements general unnesting, which decorrelates arbitrary queries and removes dependent joins from the optimized plans. The CTEs also capture the join order, including the build and probe sides of the joins. To avoid a second join reordering by the target system, we force it to follow the join order of the optimized query plan using system-specific settings and hints where possible. SQL-to-SQL optimization also enables the target systems to further optimize queries and adapt them to their specific execution and storage engines.

Figure 2 depicts a query on the TPC-DS `item` table. While the query (on the left) is easy to write and understand for humans or LLMs, its naive execution leads to quadratic runtime due to the correlated subquery. Conceptually, the subquery is evaluated for every tuple of the outer query, but the actual data allows for a more efficient evaluation in which the subquery is evaluated only once for each combination of its free variables. To achieve that, the Umbra query optimizer translates the query into relational algebra, automatically decorrelates it using its general unnesting implementation, and applies further optimizations. The resulting query plan is then translated back into SQL as an operator-oriented query, with one CTE per operator. Our implementation supports the SQL-to-SQL translation of all Umbra plan operators into the SQL dialects of ClickHouse, DuckDB, PostgreSQL, and SQL Server.

To demonstrate the advantage of this approach, we evaluate the *human-written* and the optimized *operator-oriented* query from Figure 2 in ClickHouse, DuckDB, PostgreSQL, and SQL Server: Table 1 shows the execution times of the original and optimized queries on the four systems and Umbra. For systems that do not implement unnesting (ClickHouse and PostgreSQL), executing this query takes considerable time, even though the `item` table has only 18,000 entries. Although DuckDB and SQL Server both unnest the query, Umbra's query plan further improves the execution times. To our surprise, ClickHouse (version 25.11) returns the wrong result for the original query; however, running the optimized version produces the correct result. The simplified CTE representation does not trigger the incorrect code path in ClickHouse.

**Statistics.** The key to finding a good query plan is accurate statistics. Umbra relies on HyperLogLog sketches for distinct count estimation, AMS sketches for join selectivity, and samples for filter selectivity. As QueryBrew optimizes queries for other databases, it cannot collect statistics when data is inserted or updated. Nevertheless, the required statistics can be computed in SQL, using native hash functions, aggregations, and simple arithmetic operations. Hence, through SQL, we achieve a full decoupling of the optimizer from the query engine and storage: A perfect match for today's open table formats and multi-engine landscape.

**Interface.** QueryBrew provides a user interface for exploring and testing optimized query plans. It allows running queries on multiple database systems, comparing their results and runtimes, and loading Umbra's optimized query plan. It consists of the following components: (1) a unified query editor and schema viewer, (2) a per-system editor that allows adapting and optimizing queries for individual systems, (3) a result view, and lastly (4) a query plan view with statistics. Figure 3 shows the full user interface.

**Table 1: Runtimes on the original correlated query from Figure 2 versus Umbra's optimized query plan on TPC-DS scale factor 1 GB.**

| | Original | | Optimized Query | | |
|---|---|---|---|---|---|
| | time | correct | time | correct | speedup |
| PostgreSQL | 40.8 s | ✔ | 2.1 s | ✔ | **19.9**× |
| ClickHouse | 4.4 s | ✖ | 0.2 s | ✔ | **21.8**× |
| SQL Server[1] | – | ✔ | – | ✔ | **2.84**× |
| DuckDB | 99 ms | ✔ | 12 ms | ✔ | **8.07**× |
| Umbra | 18 ms | ✔ | 18 ms | ✔ | **1.00**× |

The goal of our demonstration is to show that query optimization through SQL is feasible and that SQL-to-SQL optimizations integrate well with existing database systems. The QueryBrew interface allows users to specify their own queries and compare the original and optimized versions. The query plan view, in particular, gives insight into potential optimizations. It can be used to explore the differences between the original and optimized query plans, such as more efficient join orders or the advantage of unnesting by eliminating dependent joins.

While the CTE-based representation gives a structured and simplified representation of the query, it can lead to performance degradation in some systems, as potential optimizations are missed by the target system's optimizer. In our experiments, we observe speedups larger than 100× and only minor slowdowns in comparison (<5×) as Umbra's optimizer improves the asymptotic complexity of the queries, whereas alternative execution strategies mostly result in a constant overhead. In such cases, users can run the original query and avoid slowdowns caused by the CTE-based representation.

## 3 DEMONSTRATION PROPOSAL

In our demonstration, visitors can explore different optimizations that Umbra applies to SQL queries. Original and optimized queries can be executed on four different database systems (ClickHouse, DuckDB, PostgreSQL, and SQL Server). Our interface supports comparing runtimes and query plans and displays query results to validate the correctness of the optimizations. We preload four well-known analytical benchmarks (TPC-H, TPC-DS, SSB, and JOB), and users can run both existing and custom queries for these datasets.

To illustrate the potential of QueryBrew, we present three scenarios in this section that demonstrate how query execution improves through unnesting and join reordering. Additionally, we explore the potential of this approach for SQL dialect translation.

**Scenario 1: Unnesting arbitrary queries.** While general unnesting can decorrelate arbitrary queries, only a few systems integrate the full algorithm as proposed by Neumann and Kemper. Implementing this optimization is non-trivial and requires substantial engineering effort. Through QueryBrew's SQL-to-SQL optimizations, unnesting is available to all database systems. We invite visitors to test this algorithm and explore the difference between systems that offer this optimization (Umbra, DuckDB, SQL Server) and systems that do not (PostgreSQL, ClickHouse).

[1] Due to SQL Server's DeWitt clause, we do not report absolute times.

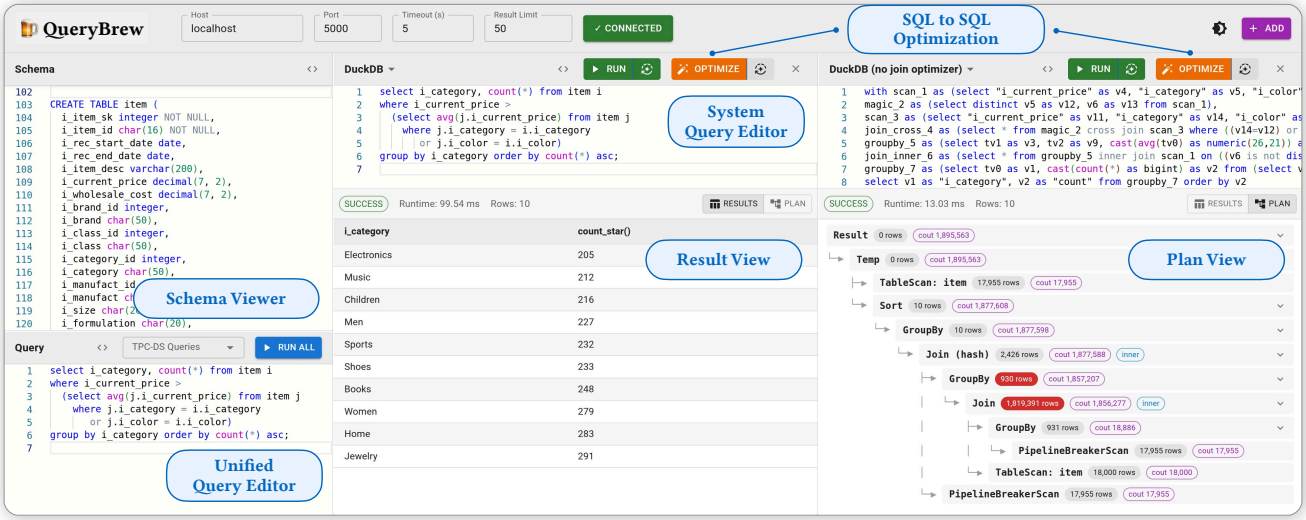


**Figure 3: QueryBrew interface**

**Scenario 2: Identifying optimal join orders.** Finding a *good* join order for complex queries is difficult and requires robust cardinality estimation. Choosing the wrong order can heavily impact the size of intermediate results and therefore query execution time. For some queries of JOB, we observe runtime improvements of more than one order of magnitude (e.g., on ClickHouse or SQL Server). Our demonstration allows users to compare Umbra's join order with that of other systems.

**Scenario 3: SQL Dialect Translation.** SQL dialects differ across systems; as a side effect of SQL-to-SQL optimization, QueryBrew also supports translating queries from the PostgreSQL dialect to other dialects. The user can specify the target dialect, and the operator-oriented query is generated to be compatible with the target system. Examples of dialect-specific syntax include the missing boolean type in SQL Server and the renaming of functions for ClickHouse and SQL Server. Unlike other SQL converters, such as SQLGlot [9], QueryBrew benefits from Umbra's optimizations and translates arbitrarily complex queries.

## 4 CONCLUSION

QueryBrew is a system-agnostic SQL-to-SQL optimization service that leverages Umbra's optimizer to exploit state-of-the-art optimization techniques. Input queries are transformed into relational algebra, then simplified, unnested, and reordered, and finally translated back into an operator-oriented SQL query. The operator-oriented query can then be executed canonically by the target system. SQL-to-SQL rewriting gives the target system a simpler and more structured representation of the input query without changing its semantics. This helps query optimizers discover more efficient plans even when they lack some advanced optimizations.

QueryBrew supports the dialects of various systems (SQL Server, ClickHouse, DuckDB, and PostgreSQL) and matches their syntax and semantic subtleties. Using this SQL-to-SQL optimization, we observe speedups by more than one order of magnitude on standard benchmarks. Our demo features an easy-to-use website that lets users explore and analyze SQL-to-SQL optimization and benchmark the original and optimized queries across different target systems.

## REFERENCES


[1] 2021. *Substrait.* Retrieved March 1, 2026 from https://github.com/substrait-io/substrait

[2] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *SIGMOD Conference*. ACM, 331–343.

[3] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. *Proc. VLDB Endow.* 16, 11 (2023), 2911–2924.

[4] Ilaria Battiston, Kriti Kathuria, and Peter Boncz. 2024. OpenIVM: a SQL-to-SQL Compiler for Incremental Computations. In *SIGMOD Conference*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 516–519.

[5] Altan Birler and Thomas Neumann. 2025. Efficient Enumeration of the Complete Join Search Space. In *Proceedings of the 19th International Symposium on Database Programming Languages*. ACM, 1–12.

[6] Nicolas Bruno, César A. Galindo-Legaria, Milind Joshi, Esteban Calvo Vargas, Kabita Mahapatra, Sharon Ravindran, Guoheng Chen, Ernesto Cervantes Juárez, and Beysim Sezgin. 2024. Unified Query Optimization in the Fabric Data Warehouse. In *SIGMOD Conference Companion*. ACM, 18–30.

[7] Haralampos Gavriilidis, Lennart Behme, Christian Munz, Varun Pandey, and Volker Markl. 2025. CompoDB: A Demonstration of Modular Data Systems in Practice. In *EDBT*. OpenProceedings.org, 1094–1097.

[8] Jie Liu and Barzan Mozafari. 2024. GenRewrite: Query Rewriting via Large Language Models. *CoRR* abs/2403.09060 (2024).

[9] Toby Mao. [n. d.]. *SQLGlot.* Retrieved March 1, 2026 from https://github.com/tobymao/sqlglot

[10] Thomas Neumann. 2025. Improving Unnesting of Complex Queries. In *BTW (LNI, Vol. P-361)*. Gesellschaft für Informatik e.V., 25–47.

[11] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[12] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI, Vol. P-241)*. GI, 383–402.

[13] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *SIGMOD Conference*. ACM, 677–692.

[14] Yuanyuan Tian. 2025. Query Optimization in the Wild: Realities and Trends. *CoRR* abs/2510.20082 (2025).

[15] Yuanyuan Tian, Jesús Camacho-Rodríguez, Carlo Curino, César A. Galindo-Legaria, Ashit Gosalia, Brian Kroth, Sergiy Matusevych, Nicolas Bruno, Ashvin Agrawal, Stefan Grafberger, Beysim Sezgin, Milan Potocnik, Mahesh Behera, Milind Joshi, and Xiaoyu Li. 2025. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Platform: Can One QO Rule Them All?. In *CIDR*. www.cidrdb.org.