QueryTorque: Swarm-of-Reasoners for Training-Free SQL Query Optimization

via Competitive Multi-Worker Generation

[Authors TBD]
February 2026 -- Paper Stub / Working Draft

[TODO: #1]

Abstract

SQL query optimization through rewriting is critical for analytical workload performance, yet existing approaches face fundamental trade-offs. Rule-based systems (LearnedRewrite, R-Bot) are constrained by fixed rule vocabularies and cannot express structural rewrites like CTE isolation or scan consolidation. Training-based methods (E^3-Rewrite) require expensive reinforcement learning pipelines and are evaluated on a single database engine. We present QueryTorque, a training-free SQL optimization system that uses a _swarm-of-reasoners_ architecture: multiple specialized LLM workers, each guided by different verified gold examples and engine-specific constraints, compete to produce the best rewrite for a given query.

With a single architecture and zero retraining, achieves a 37% win rate on both DuckDB (TPC-DS) and PostgreSQL (DSB) -- the first system to demonstrate cross-engine generalization for LLM-based query rewriting. On TPC-DS SF10 under DuckDB, QueryTorque achieves speedups of up to 6.28x (Q88) with 17 winning queries and an average speedup of 1.19x across 43 validated queries. On DSB SF5 under PostgreSQL, QueryTorque achieves 19 wins across 52 queries, recovering two timeout queries (300 s->68 ms and 300 s->766 ms) and producing up to 122x speedup on non-timeout queries. The largest single improvement, Q092 (4,428x), results from a compositional three-pattern rewrite discovered by the swarm. Unlike prior work, QueryTorque (1) requires no model fine-tuning, instead leveraging frontier reasoning models at inference time; (2) represents queries as logical-block DAGs with node contracts and cost attribution; (3) decomposes complex benchmark queries into typed sub-problems (multi-CTE, aggregation, join-path) that expose different optimization surfaces; (4) learns bidirectionally from both successful rewrites and validated regressions; (5) operates across multiple database engines with engine-aware constraint injection; and (6) employs a stateful multi-iteration pipeline where winners are promoted as baselines for subsequent rounds.

Introduction

Efficient query execution is central to modern analytical database systems. SQL query rewriting -- transforming a query into a semantically equivalent but faster form -- has been studied extensively through rule-based systems[zhou2021learned, calcite2018], heuristic optimizers[graefe1993volcano], and more recently through large language model (LLM) approaches[rbot2024, e3rewrite2026, llmr2_2024].

Despite significant progress, existing methods face three fundamental limitations:

Limited expressiveness. Rule-based systems (Apache Calcite, PostgreSQL's optimizer, LearnedRewrite[zhou2021learned]) operate over a fixed vocabulary of predefined transformations. They cannot express rewrites that span multiple query blocks, such as isolating dimension tables into filtered CTEs, consolidating repeated table scans into single-pass CASE aggregations, or decomposing self-joins -- strategies that yield 2--6x speedups on real workloads.

Expensive training pipelines. E^3-Rewrite[e3rewrite2026] addresses expressiveness by training LLMs via reinforcement learning (GRPO) with execution-aware rewards. However, this requires (1) a two-stage curriculum over thousands of training queries, (2) repeated query execution for reward computation, and (3) retraining for each new database engine or schema. The resulting models are also frozen at training time and cannot incorporate new optimization patterns discovered at deployment.

Single-engine, single-strategy design. Both R-Bot[rbot2024] and E^3-Rewrite target PostgreSQL exclusively. Their architectures assume a fixed execution environment and do not account for the significant behavioral differences between engines (e.g., DuckDB's columnar execution vs.\ PostgreSQL's row-oriented model, CTE materialization semantics, OR-to-UNION effectiveness). No prior system has demonstrated that a single architecture generalizes across engines without retraining.

We present QueryTorque, a training-free SQL optimization system that addresses all three limitations through six key design decisions:

* **Logical-block DAG representation.** Rather than injecting raw EXPLAIN plans (physical operators), we parse the SQL into a _Query DAG_ where nodes are the semantic blocks that rewrites operate on -- CTEs, subqueries, and the main query. Each node carries a _contract_ (output columns, grain, required predicates), _downstream usage_ (which columns consumers actually reference), and _cost attribution_ (percentage of total estimated cost). This provides the LLM with structural context at the same granularity as the rewrite, enabling cross-node transformations (new CTEs, filter pushdown

across node boundaries) that physical-plan representations cannot express.

* **Swarm-of-reasoners architecture.** Rather than relying on a single LLM call, QueryTorque dispatches multiple specialized workers in parallel, each seeded with different verified gold examples targeting distinct optimization strategies (e.g., decorrelation, CTE isolation, scan consolidation). Workers compete, and only the best validated candidate is selected. This produces higher coverage than single-shot generation while requiring no training.

* **Bidirectional learning from examples.** The prompt includes both _gold examples_ (rewrites with verified speedups) and _regression warnings_ (rewrites with verified slowdowns and their anti-pattern explanations). This teaches the model not only what to do but what to avoid -- a signal absent from prior systems.

* **Query-type decomposition.** For complex benchmark queries, we decompose each query into typed sub-problems -- _multi-CTE_ (hierarchical CTE pipelines), _aggregation_ (GROUP BY with conditional branches), and _join-path_ (SPJ validation variants) -- that expose different optimization surfaces. The same underlying business logic is formulated in structurally distinct ways, allowing the swarm to attack each formulation independently and select the best result per type.

* **Engine-aware constraint injection.** A library of severity-tiered constraints, many auto-generated from observed regressions, is injected using a "sandwich" attention pattern (critical constraints at both the start and end of the prompt). Constraints are engine-specific: for example, OR-to-UNION is encouraged for DuckDB but prohibited for PostgreSQL, where it causes regressions.

* **Stateful multi-iteration promotion.** QueryTorque operates as a state machine: winning rewrites from round N become the baseline for round N+1, enabling compositional optimization where later rounds build on earlier gains.

[TODO: Add figure: system overview diagram showing swarm architecture with DAG analysis, worker dispatch, validation, and promotion flow.]

Related Work

Heuristic and rule-based rewriting.
Traditional query optimizers apply rewrite rules in fixed or heuristically explored orders. PostgreSQL[postgresql] uses a sequential rule pipeline, while Volcano[graefe1993volcano] and Cascades[graefe1995cascades] explore rule combinations via branch-and-bound search. Apache Calcite[calcite2018] provides an

extensible framework with 70 built-in rules.
LearnedRewrite[zhou2021learned] applies Monte Carlo Tree Search (MCTS) guided by learned cost models to search over Calcite's rule space, but remains bounded by Calcite's rule vocabulary and requires schema-specific cost model training.

LLM-augmented rule selection.
LLM-R^2[llmr2_2024] prompts GPT-3.5 with demonstrations to select Calcite rules. R-Bot[rbot2024] extends this with multi-source evidence

prepar

Overflow), hybrid structure-semantics retrieval, and step-by-step LLM-guided rule arrangement with reflection. While R-Bot achieves strong results and is deployed at Huawei, it fundamentally operates within Calcite's rule vocabulary -- it _selects_ rules but cannot _generate_ novel rewrites. For example, it cannot express CTE isolation, scan consolidation, or dimension prefetching, which are among our most effective transforms.

LLM-based direct rewriting.
E^3-Rewrite[e3rewrite2026] fine-tunes Qwen/LLaMA models via GRPO with a three-component reward (executability, equivalence, efficiency) and a two-stage curriculum. It achieves 25.6% latency reduction on TPC-H and 99.6% equivalence rate. E^3-Rewrite injects linearized EXPLAIN plans as "execution hints," exposing physical operators (Seq Scan, Hash Join) and their costs. While useful for identifying bottleneck operators, this physical-level representation does not expose the logical block structure (CTE boundaries, subquery scopes, cross-node data contracts) that guides structural rewrites. Furthermore, the approach requires (1) significant compute for RL training, (2) access to query execution for reward computation during training, and (3) retraining for new engines.
QUITE[quite2025] is the closest prior work to QueryTorque: it is training-free, uses multiple LLM agents (a rewriter, a corrector, and a hint injector), and targets free-form SQL rewriting without a rule engine. However, QUITE differs in several key respects: (1) it uses query hints (e.g., `SET` commands, optimizer directives) as a primary mechanism, which are engine-specific and do not change the SQL structure;
(2) it lacks a systematic learning loop from deployment feedback;
(3) it does not use a structured query representation (DAG or otherwise) to guide the rewriter; and (4) it reports results on PostgreSQL only, with no cross-engine evaluation.

Positioning of .
Table [tab:comparison] positions QueryTorque against prior systems. QueryTorque is the first system that combines training-free operation, unrestricted rewrite expressiveness (no rule engine), multi-engine support, and bidirectional learning from both successes and failures.

Table: Comparison of SQL rewriting systems.

-- -

| |

-- -

Train

Rule engine required | Yes | Yes | No | No
Multi-engine | No | No | No | Yes
Rewrite expressiveness | Calcite rules | Calcite rules | Free-form | Free-form
Query representation | Cost | Rule^ | Phys.\ plan | Logic.\ DAG
Workers / candidates | 1 | 1 | N | 4+
Regression learning | No | No | No | Yes
Constraint injection | No | No | No | Yes
Stateful iteration | No | Yes* | No | Yes
-- -

^-Bot uses one-hot encoding of matched Calcite rule specifications.

*R-Bot's reflection loop reruns rule selection; it does not promote winning SQL as a new baseline.

System Overview

QueryTorque processes a SQL query through a five-phase pipeline (Figure [fig:overview]), orchestrated by a state machine that supports multi-round optimization.

[TODO: Figure [fig:overview]: End-to-end system diagram.]

-- Phase 1: Structural Analysis via Query DAG --

A central design choice in QueryTorque is the level of abstraction at which we represent query structure. E^3-Rewrite injects raw EXPLAIN plans -- physical operator trees showing `Seq Scan`, `Hash Join`, `Gather` nodes with cost estimates and row counts. R-Bot represents queries as one-hot vectors of matched Calcite rule specifications. We argue that neither representation aligns with the granularity at which SQL rewrites actually operate.

SQL optimizations -- decorrelating a subquery, isolating a dimension into a CTE, pushing a filter across a CTE boundary, consolidating repeated scans -- operate at the level of _logical query blocks_: CTEs, subqueries, and the main SELECT. A developer does not rewrite a Hash Join node; they restructure the CTE that feeds it. We therefore parse the input SQL into a _Query DAG_ that mirrors this granularity.

DAG construction.
Given a SQL query q, we parse it via `sqlglot`[sqlglot] and construct a DAG G = (V, E) where:

* Each node v V represents a logical block: a CTE definition, a

subquery, or the main query body.

* Each edge (u, v) E represents a data dependency: node v references the output of node u (e.g., the main query references a CTE).

Each node is annotated with _structural flags_ detected via AST traversal: `GROUP\_BY`, `WINDOW`, `UNION\_ALL`, `CORRELATED`, `IN\_SUBQUERY`. These flags serve as lightweight pattern indicators that help the LLM identify optimization opportunities without parsing the SQL itself.

Node contracts.

For each node v, we compute a _contract_ -- the interface between v and its consumers:

* **Output columns**: The column names this node produces (extracted from the SELECT clause).
* **Grain**: The GROUP BY keys, if any -- defining the aggregation level of the node's output.
* **Required predicates**: WHERE and HAVING conditions that must be preserved for semantic equivalence.

Contracts make explicit what each node promises to its consumers. This is critical for safe cross-node rewrites: if a CTE's contract specifies `output\_columns = [customer\_id, total\_return]`, a rewrite that restructures the CTE must preserve these columns or risk breaking downstream references.

Downstream usage analysis.

For each node v, we compute which of its output columns are _actually referenced_ by consumer nodes. If a CTE outputs 12 columns but only 3 are used downstream, the LLM can safely project away the unused 9 -- a common source of speedup in columnar engines like DuckDB where eliminating column reads reduces I/O.

Cost attribution.

When an EXPLAIN plan is available (either from the target engine directly or from a cached plan archive), we map physical operators back to their logical DAG nodes and compute the percentage of total estimated cost attributable to each node. When EXPLAIN is unavailable, we fall back to heuristic cost splitting based on node complexity (number of joins, presence of GROUP BY, etc.). The per-node cost profile is rendered as inline comments in the DAG topology section of the prompt:

"`
Query Structure (DAG)

Nodes:
customer_total_return [CTE]
tables: catalog_returns, date_dim,

customer_address
flags: GROUP_BY
cost: 62
grain: [ctr_customer_sk, ctr_state]
output: [ctr_customer_sk, ctr_state,
ctr_total_return]

main_query [MAIN]
refs: customer_total_return
tables: customer_total_return,
customer_address, customer
flags: CORRELATED
cost: 38

Edges:
customer_total_return -> main_query

Semantic intents (optional).

For complex queries, we optionally pre-compute _semantic intents_ -- natural-language descriptions of what each DAG node computes (e.g., ``computes per-state average return amount for catalog customers''). These are generated once per query via a lightweight LLM call and cached. When present, they are attached to their respective nodes in the prompt, giving the rewriter human-readable context alongside the structural metadata.

Knowledge base patterns.

The prompt also includes engine-specific _knowledge base patterns_ that explain _why_ certain rewrites work at the optimizer level. For example, the pattern `SQL-DUCK-014` (Grouped Top-N via Window vs.\ LATERAL) explains that DuckDB cannot automatically transform a `ROW\_NUMBER() OVER (PARTITION BY ...)` into a LATERAL subquery because the semantic rewrite is not cost-based. This knowledge -- the _optimizer's limitations_ -- is absent from EXPLAIN plans and rule specifications, yet is precisely what guides productive rewrites.

Comparison with prior representations.

Table [tab:repr_comparison] compares the three approaches.

Table: Query representation for LLM-based rewriting.

p1.5cmp1.5cmp1.5cm
-- -

Property & **E^3 (Phys.)** & **R-Bot (Rule)** & **Ours (DAG)**
-- -

Granu

```
Cross-node rewrites  |  No  |  No  |  Yes
Contract tracking  |  No  |  No  |  Yes
Engine-independent  |  No  |  Calcite  |  Yes
Cost attribution  |  Per-op  |  No  |  Per-block
```

```
Unused col.\ detection  |  No  |  No  |  Yes
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
```

The key advantage is **alignment**: the DAG represents the query at the same level of abstraction at which rewrites happen, giving the LLM a structural map rather than a physical execution trace.

-- Phase 2: Example Retrieval --

We maintain a curated library of gold examples -- pairs of (original SQL, optimized SQL) with verified speedups -- organized by database engine. Each example is annotated with the transform pattern it demonstrates (e.g., `date\_cte\_isolate`, `decorrelate`, `single\_pass\_aggregation`).

For a given query q, we retrieve the top-k most relevant examples using tag-based overlap matching:

* **Tag extraction.** We extract tags from q: table names, SQL keywords (GROUP BY, HAVING, EXISTS, etc.), and structural patterns (correlated subquery, self-join, window function).
* **Category classification.** We assign q to an archetype (filter-pushdown, aggregation-rewrite, set-operations, general) based on its structural features.
* **Overlap ranking.** We rank examples by the number of shared tags with q, filtered by database engine.

This approach is deliberately lightweight -- no embedding models, no FAISS indexes, no GPU inference. Despite its simplicity, it achieves effective retrieval because the tag vocabulary is domain-specific to SQL optimization patterns.

In addition to gold examples, we retrieve _regression examples_: queries where specific transforms caused verified slowdowns. These are injected as anti-pattern warnings, teaching the model what to avoid for structurally similar queries.

-- Query-Type Decomposition --

A distinctive feature of QueryTorque's evaluation pipeline is the decomposition of complex benchmark queries into _typed sub-problems_. Each base query (e.g., DSB query 092) is formulated in up to three structurally distinct variants:

* **Multi-CTE (`\_multi**)`: The canonical complex form with hierarchical CTE pipelines, nested subqueries, and multi-level data dependencies. These queries exercise the full range of structural rewrites: decorrelation, CTE isolation, filter pushdown across node boundaries.

* **Aggregation (`\_agg**)`: Reformulations emphasizing GROUP BY operations with conditional OR branches and aggregate functions (AVG, SUM, COUNT). These expose optimization surfaces for scan consolidation, aggregate pushdown, and OR-to-UNION decomposition.
* **Join-path (`\_spj\_spj**)`: Select-Project-Join validation variants with identical WHERE clauses and join structures but simplified SELECT lists (typically MIN of key columns). These test join reordering and dimension-filter-first strategies without aggregation overhead.

This decomposition serves two purposes. First, different query formulations expose different optimization opportunities: a query whose `\_multi` variant resists optimization may yield significant speedups in its `\_agg` formulation because the flatter structure enables scan consolidation. In our PostgreSQL DSB evaluation, 10 of 19 wins came from `\_multi` variants, 3 from `\_agg`, and 6 from `\_spj\_spj`, confirming that no single formulation captures all optimization potential.

Second, the swarm processes each variant independently, allowing different workers to specialize on different structural patterns. The best result per base query is selected across all variants and all workers, maximizing coverage. This approach is orthogonal to the swarm architecture and could benefit any LLM-based rewriting system.

-- Phase 3: Prompt Construction --

The prompt assembles eight attention-ordered sections:

* **Role framing**: System instruction for SQL optimization.
* **Original SQL**: Pretty-printed with query identifier.
* **DAG topology**: Nodes, edges, and per-node cost percentages.
* **Constraints**: Severity-tiered rules in sandwich pattern (CRITICAL at top and bottom, HIGH/MEDIUM in middle).
* **History**: Previous attempts with status, speedup, and transforms (for multi-iteration mode).
* **Gold examples**: Top-k matched examples with original -> optimized SQL pairs and speedup annotations.
* **Regression warnings**: Anti-pattern examples for similar query structures.
* **Output format**: Return complete rewritten SQL with summary.

The sandwich constraint pattern exploits the well-documented primacy and recency bias in LLM attention[liu2024lost], ensuring critical constraints (semantic equivalence, literal preservation, column completeness) receive maximum attention.

-- Phase 4: Multi-Worker Generation --

This is the core differentiator of QueryTorque. Rather than generating a

single candidate, we dispatch W workers in parallel (default W=4), each receiving a _different_ set of gold examples targeting a distinct optimization strategy:

* **Worker 1** (Restructure): decorrelate, pushdown, early_filter
* **Worker 2** (CTE Isolation): date_cte_isolate, dimension_cte_isolate, multi_date_range_cte
* **Worker 3** (Prefetch): prefetch_fact_join, multi_dimension_prefetch, materialize_cte
* **Worker 4** (Consolidate): single_pass_aggregation, or_to_union, intersect_to_exists, union_cte_split

Each worker calls a frontier reasoning model (e.g., DeepSeek-Reasoner[deepseek_r1], OpenAI o1[openai_o1]) with the full prompt augmented by its strategy-specific examples. The use of reasoning models is deliberate: SQL optimization requires multi-step logical deduction (identifying bottlenecks, planning transformations, verifying equivalence), which benefits from chain-of-thought reasoning[deepseek_r1].

All W candidates are collected and passed to Phase 5.

-- Phase 5: Validation and Selection --

Each candidate undergoes a three-stage validation:

* **Syntax gate**: Parse with `sqlglot`; reject unparseable candidates.
* **Semantic equivalence**: Execute both original and rewritten queries; compare row counts and MD5 checksums of result sets.
* **Performance measurement**: Execute the rewritten query using our trimmed-mean protocol (Section [sec:validation]).

The candidate with the highest validated speedup is selected. If no candidate achieves speedup 1.0, the original query is preserved ("do no harm" principle).

-- State Machine and Promotion --

QueryTorque supports multi-round optimization via a state machine. After round N:

* Queries with speedup (default = 1.05) are _promoted_: their optimized SQL becomes the baseline for round N+1.
* Non-winners retain their original baseline.
* A _promotion context_ -- a natural-language summary of what worked and why -- is generated and injected into the next round's prompt as history.

This enables compositional optimization: round 1 might decorrelate a subquery, and round 2 might then isolate a dimension CTE that was previously buried inside the correlated block.

Constraint and Learning System

-- Engine-Aware Constraints --

QueryTorque maintains a library of constraints -- rules that restrict the LLM's rewriting behavior based on verified failure modes. Each constraint specifies:

* **Severity**: CRITICAL, HIGH, MEDIUM, LOW
* **Engine scope**: Which database engines it applies to
* **Description**: Natural-language explanation of the anti-pattern
* **Provenance**: The query and regression that motivated the constraint

Constraints are loaded from a JSON directory and injected into every prompt. Examples of constraints:

* `or\_to\_union\_limit` (HIGH, DuckDB): Limit OR->UNION to <=3 branches. _Motivation_: Q13 and Q48 showed 0.23--0.41x regressions from 9-branch UNION rewrites that caused 9x fact table scans.
* `or\_to\_union\_harmful` (CRITICAL, PostgreSQL): Do not apply OR->UNION on PostgreSQL. _Motivation_: PostgreSQL's optimizer already handles OR efficiently; the rewrite removes the optimizer's ability to merge scans.
* `no\_materialize\_exists` (CRITICAL, all): Do not materialize EXISTS subqueries into CTEs. _Motivation_: Q16 showed 0.14x regression when a correlated EXISTS was hoisted into a materializing CTE, breaking the short-circuit evaluation.

Currently, QueryTorque has 11 general constraints plus 5 PostgreSQL-specific and 7 DuckDB-specific constraints, for a total of 23.

-- Bidirectional Learning --

After each optimization round, the system records a structured `LearningRecord` capturing:

* **What was tried**: examples recommended, transforms suggested
* **What happened**: status (WIN/IMPROVED/NEUTRAL/REGRESSION/ERROR), speedup ratio, all error messages, error category (syntax|semantic|timeout|execution)
* **Effectiveness scores**: per-example and per-transform success

rates

These records feed four feedback loops:

* **Example pool evolution**: New winners become gold example candidates; weak examples (no wins in 3+ batches) are retired.
* **Constraint auto-generation**: Clusters of regressions with common structural patterns produce new constraint JSON files.
* **Strategy leaderboard**: Per-archetype, per-transform success rates guide the analyst layer's strategy recommendations.
* **Global knowledge injection**: Aggregate statistics (pattern effectiveness, known regressions) are included in prompts as "global learnings."

Experimental Evaluation

-- Setup --

Benchmarks.
We evaluate on two standard decision-support benchmarks:

* **TPC-DS** (SF1, SF10): 88 query templates exercising complex joins, correlated subqueries, window functions, and set operations. Executed on **DuckDB** v1.1[duckdb2019].
* **DSB** (SF5, SF10): 52 queries adapted from TPC-DS for modern decision-support workloads[dsb2021]. Executed on **PostgreSQL** v16.

LLM backend. DeepSeek-Reasoner (deepseek-reasoner) as the primary reasoning model. We also evaluate with Kimi K2.5 via OpenRouter. No model fine-tuning is performed.

Hardware. WSL2 on Windows 11, Intel Core i7, 32GB RAM, DuckDB databases on NVMe SSD.

Baselines. We compare against:

* **Original**: Unmodified query execution
* **LLM-only (GPT-4o)**: Direct prompting without examples, constraints, or multi-worker architecture
* **R-Bot**: LLM-guided Calcite rule selection with evidence retrieval and reflection[rbot2024]
* **E^3-Rewrite**: RL-trained LLM rewriting with execution hints[e3rewrite2026]

[TODO: Run R-Bot and E^3-Rewrite baselines on our TPC-DS/DSB setup, or
cite their published numbers on overlapping benchmarks.]

-- Validation Protocol --

We use two validation protocols, applied consistently across all methods:

* **3-run mean**: Run 3 times, discard the first run (warmup),
average the last 2. Used for rapid iteration.
* **5-run trimmed mean**: Run 5 times, remove the minimum and
maximum, average the remaining 3. Used for final reported numbers.

A query is classified as:

* **WIN**: speedup 1.10
* **IMPROVED**: speedup [1.05, 1.10)
* **NEUTRAL**: speedup [0.95, 1.05)
* **REGRESSION**: speedup < 0.95

Semantic equivalence is verified by comparing row counts and MD5 checksums
of full result sets between original and rewritten queries.

-- Main Results: DuckDB TPC-DS SF10 --

Table: DuckDB TPC-DS SF10 results (43 validated queries, 4 workers).

-- -

Status & **Count** & **%** & **Avg Speedup**
-- -

WIN (

```
PASS (0.95--1.10x)  |  17  |  39%  |  1.02x
REGRESSION (<0.95x)  |  9  |  21%  |  0.78x
```

-- -

Overall & 43 & & **1.19x**
-- -

Table [tab:duckdb_results] summarizes results across 43 validated
queries after the 4-worker swarm. The system achieves a 39% win rate
with an average speedup of 1.19x across all queries. Among winners,
the average speedup is 1.94x.

Top winners.

* Q88: **6.28x** via `or\_to\_union` -- converting a
complex 8-way OR filter on `time\_dim` into a 4-branch UNION ALL
with pre-filtered time buckets.

* Q9: **4.47x** via `single\_pass\_aggregation` --
consolidating 10 repeated scans of `store\_sales` into a single
scan with CASE-based conditional aggregation.
* Q40: **3.35x** via `multi\_cte\_chain` -- isolating
three date dimension joins into pre-filtered CTEs.
* Q46: **3.23x** via `triple\_dimension\_isolate`.
* Q42: **2.80x** via `dual\_dimension\_isolate`.

Most effective transform pattern.
`date\_cte\_isolate` -- pre-filtering `date\_dim` into a CTE
before joining with fact tables -- produced 12 wins at 1.34x average
speedup. This pattern is not expressible in Apache Calcite's rule
vocabulary, illustrating the advantage of unrestricted LLM-based rewriting.

Scale factor correlation.
We observe a Pearson correlation of r=0.77 between SF1 and SF10
speedups, confirming that SF1 is a reliable proxy for rapid experimentation.
Winners tend to _scale better_: Q88 improved from 1.02x at SF1
to 6.28x at SF10.

-- Main Results: PostgreSQL DSB SF5 --

Table: PostgreSQL DSB SF5 results (52 queries, 6 workers, 3 iterations).

-- -

Status & **Count** & **%** & **Avg Speedup**
-- -

WIN (>

IMPROVED ([1.05,1.10)) | 16 | 31% | 1.43x
NEUTRAL ([0.95,1.05)) | 11 | 21% | 1.03x
REGRESSION (<0.95x) | 4 | 8% | 0.72x
ERROR | 2 | 4% | --
-- -

Two WINs (Q032, Q092) are timeout recoveries (300 s baseline).
Excluding these, average non-timeout WIN speedup is 15.8x.

Table [tab:pg_results] shows results on PostgreSQL after the full
6-worker, 3-iteration swarm. The 37% win rate matches DuckDB's 39% -- achieved
with the _same architecture, same gold examples library, and zero
retraining_. Only the constraint set differs between engines.

Top winners (non-timeout).
Q081 (122x: 257 s->2.1 s), Q010 (73x: 2.9 s->39 ms),
Q023 (66x: 7.5 s->113 ms), Q013_agg (53x:
2.8 s->52 ms).

Timeout recoveries.
Q032 and Q092 both time out at 300 s in their original form. The swarm recovers Q032 to 766 ms and Q092 to 68 ms. Both use the same compound pattern: `decorrelate` + `date\_cte\_isolate` + `dimension\_cte\_isolate`. We analyze Q092 in detail in Section [sec:case_study].

Engine-specific findings. Several transforms that are effective on DuckDB are harmful on PostgreSQL:

* `or\_to\_union`: Produces regressions on PostgreSQL (the optimizer already handles OR predicates efficiently).
* `CTE materialization`: PostgreSQL's CTE materialization fence is double-edged -- it can prevent the optimizer from pushing predicates into CTEs, causing regressions.

These findings motivated our engine-specific constraint system.

-- Case Study: Q092 Compositional Rewrite --

Q092 is a DSB query with a correlated subquery that compares each customer's web return amount against a per-state average, joined through `date\_dim`, `customer\_address`, and `web\_returns`.
The original query times out at 300 s on PostgreSQL SF5 -- the correlated subquery forces a nested-loop execution plan that re-scans `web\_returns` for every outer row.

The swarm dispatched 4 workers. Three produced valid rewrites:

* **W1** (conservative): Decorrelated the subquery into a CTE with GROUP BY, preserving the join structure. _Result_: 35.1 s (8.6x). The decorrelation eliminated nested-loop rescans but left the date and address joins unoptimized.
* **W3** (aggressive prefetch): Same decorrelation plus pre-filtering `date\_dim` into a CTE. _Result_: 15.8 s (19.0x). The date CTE reduced the join cardinality but PostgreSQL's CTE materialization fence prevented further pushdown.
* **W4** (novel elimination): Applied all three patterns compositionally -- decorrelating the subquery, isolating `date\_dim` into a pre-filtered CTE, _and_ isolating `customer\_address` with an inline state filter. _Result_: 67.7 ms (4,428x). The three-way decomposition allowed PostgreSQL to use hash joins throughout, with each dimension pre-filtered to a small cardinality before touching the fact table.

This case illustrates three properties of the swarm architecture:
(1) _compositional discovery_ -- W4's three-pattern rewrite was not explicitly programmed; it emerged from the worker's strategy seed and the reasoning model's chain-of-thought;
(2) _worker diversity_ -- the 520x gap between W1 and W4

demonstrates that a single-worker system would have left most of the performance on the table;
(3) _validation necessity_ -- all three rewrites are semantically correct, but their performance differs by orders of magnitude. Without execution-based validation, there is no way to distinguish W4's breakthrough from W1's modest improvement.

-- Worker Strategy Analysis --

Table: Per-worker win attribution (DuckDB TPC-DS SF10, 4 workers).

-- -

Worker & **Strategy Focus** & **Wins**

-- -

W1 |

W2 | date_cte_isolate, dimension_cte_isolate | 9
W3 | prefetch_fact_join, materialize_cte | 8
W4 | single_pass_agg, or_to_union | 6

-- -

Total unique wins & & **30**

-- -

Table [tab:worker_wins] shows that wins are distributed across all four workers, with no single strategy dominating. This validates the swarm-of-reasoners design: a single-worker system using any one strategy would miss 70--77% of the wins.

[TODO: Add ablation: single-worker vs.\ 2-worker vs.\ 4-worker win rates.]

-- Ablation Study --

We report two ablations supported directly by the operational data.

Single-iteration vs.\ multi-iteration.
The DuckDB TPC-DS leaderboard tracks which iteration produced each winning rewrite. Of the 43 validated queries:

Table: Wins by iteration source (DuckDB TPC-DS SF10).

-- -

Source & **Description** & **Wins** & **Cumul.**

-- -

Iter (

Iter 1 (Retry3W) | 3-worker retry on neutrals | 12 | 20
Iter 2 (Retry4W) | 4-worker retry, full swarm | 23 | 43
Iter 3 (Swarm) | 6-worker ensemble + promotion | 49 | 92

-- -

Only 8.7% of wins (8/92) came from the initial single-worker iteration. Each subsequent round nearly doubled the cumulative win count, validating the multi-iteration promotion mechanism. The improvement from Iter 0 to Iter 1 (8->20 wins) demonstrates the value of the swarm alone; the further improvement from Iter 1 to Iter 2 (20->43) shows that promotion -- using winning SQL as the new baseline -- unlocks compound optimizations that a single round cannot discover.

Worker coverage.
In the swarm batch (Iter 2--3), we tracked which worker produced the best rewrite for each query:

Table: Per-worker win attribution (DuckDB swarm batch, unique wins).

-- -

Worker & **Strategy Focus** & **Best** & **Unique**
-- -

W1 |

W2 | date/dim CTE isolate, multi_date_range | 9 | 7
W3 | prefetch_fact_join, materialize_cte | 8 | 7
W4 | single_pass_agg, or_to_union | 6 | 5
-- -

Total & & **30** & **24**
-- -

"Unique" = queries where only that worker achieved a winning speedup; no other worker found a valid improvement.

80% of winning queries (24/30) were _uniquely_ discovered by a single worker -- no other worker found a valid improvement for those queries. The best single worker (W2) captures only 30% of wins (9/30). A system using any single strategy would miss 70--83% of the optimization opportunities, directly validating the swarm-of-reasoners design.

[TODO: Additional ablations to run: without DAG topology (raw SQL only), without constraints, without gold examples, reasoning model vs.\ standard model (DeepSeek-Reasoner vs.\ GPT-4o).]

-- Comparison with Prior Work --

Direct comparison is complicated by differences in benchmark versions, scale factors, and hardware. We report each system's published numbers on overlapping benchmarks and compute aggregate statistics in the same format.

DSB on PostgreSQL (head-to-head).

E^3-Rewrite[e3rewrite2026] and R-Bot[rbot2024] both report
DSB results on PostgreSQL. We compute our aggregate latency numbers in the
same format (average across all queries, including neutrals and regressions):

Table: DSB PostgreSQL aggregate comparison.

-- -

System & **Avg Orig.** & **Avg Rewr.** & **Reduction**
-- -

R-Bot|

```
E^3-Rewrite[e3rewrite2026]  |  38.83 s  |  16.93 s  |  56.4%
QueryTorque (SF5, excl.\ TO)  |  12.90 s  |  5.09 s  |  60.5%
QueryTorque (SF5, incl.\ TO)  |  24.39 s  |  4.91 s  |  79.9%
```

-- -

TO = timeout queries (Q032, Q092 with 300 s baseline).
E^3/R-Bot numbers are SF10; ours are SF5. "Excl.\ TO" uses 48 queries;
"Incl.\ TO" uses all 50 non-error queries.

Even excluding timeout recoveries and at a smaller scale factor,
QueryTorque achieves a comparable reduction rate (60.5% vs.\ E^3's 56.4%)
without any model training.

Key qualitative differences.

* **Rewrite expressiveness**: QueryTorque discovers transforms
outside any rule vocabulary (e.g., `single\_pass\_aggregation`,
`time\_bucket\_aggregation`, `self\_join\_decomposition`).
These account for 8 of our 19 PostgreSQL wins.
* **Peak speedup**: Q092's 4,428x (timeout recovery) is
the largest single improvement reported in the literature.
E^3-Rewrite's best is 10x on timeout queries; R-Bot's
best is 99% reduction on individual Calcite queries.
* **Cross-engine generalization**: QueryTorque achieves 37% win
rates on both DuckDB and PostgreSQL with zero retraining.
E^3-Rewrite and R-Bot evaluate on PostgreSQL only.

Discussion

Why reasoning models matter.

SQL optimization requires multi-step logical reasoning: identifying
bottlenecks, planning a sequence of transformations, and mentally verifying
that the rewrite preserves semantics. Reasoning models (DeepSeek-Reasoner,
o1-style) excel at this because they allocate variable compute to problem
difficulty via chain-of-thought. Standard models (GPT-4o, Claude Sonnet)

produce more superficial rewrites that miss compositional opportunities.

[TODO: Quantify this with a head-to-head comparison.]

All discovered patterns are known.

A notable finding is that all 17 optimization patterns discovered by QueryTorque are _known_ database optimization techniques (decorrelation, predicate pushdown, CTE isolation, scan consolidation, etc.). The system does not invent novel strategies -- it rediscovers and correctly applies known techniques to specific queries where they are beneficial. This suggests that the current bottleneck is not strategy invention but strategy _selection and application_, which is precisely what swarm-of-reasoners addresses.

Limitations.

* **API cost**: 4 workers x reasoning model calls per query. At current pricing, approximately \0.10--0.50 per query optimization. Justified for slow queries (>1s) but not for sub-second queries.
* **Equivalence verification**: We verify via result comparison, not formal equivalence proofs. This is sound for the tested database instance but does not guarantee equivalence over all possible inputs.
* **Scale factor sensitivity**: Several PostgreSQL wins at SF5 degrade or regress at SF10 (e.g., Q023: 66x->1.07x), suggesting that some rewrites exploit cardinality-dependent optimizer decisions that change at larger scale.
* **Cold start**: Effective gold examples require initial benchmark runs to discover winning transforms.

Reproducibility

QueryTorque is designed for reproducibility. We release:

* **Source code**: The complete optimization pipeline including DAG construction, prompt assembly, multi-worker dispatch, validation, and learning system. Built in Python with `sqlglot`[sqlglot] for SQL parsing and AST manipulation.
* **Gold example library**: 29 DuckDB examples, 5 PostgreSQL examples, and 74 DSB catalog rules, each with original and optimized SQL, verified speedup, and transform annotations.
* **Constraint library**: 23 constraints (11 general, 5 PostgreSQL-specific, 7 DuckDB-specific) in JSON format with severity, description, and provenance.
* **Benchmark scripts**: Automated runners for TPC-DS (DuckDB) and DSB (PostgreSQL) with configurable scale factors, worker counts, and iteration depth.

* **Validation protocol**: 5-run trimmed-mean timing with semantic equivalence checking (row count + MD5 checksum comparison).
* **Leaderboard data**: Complete per-query results including original/optimized latencies, transforms applied, worker attribution, and iteration source.

The system requires only a `sqlglot` installation and API access to a reasoning-capable LLM (DeepSeek-Reasoner, OpenAI o1, or equivalent). No GPU, no model training, and no rule engine installation is needed.

Conclusion

We presented QueryTorque, a training-free SQL optimization system built on three core ideas: (1) a logical-block DAG representation that provides LLMs with structural context aligned to the granularity of rewrites; (2) a swarm-of-reasoners architecture where specialized workers compete to produce the best validated rewrite; and (3) query-type decomposition that exposes different optimization surfaces from the same underlying query. With a single architecture and zero retraining, QueryTorque achieves a 37% win rate on both DuckDB (TPC-DS, up to 6.28x) and PostgreSQL (DSB, up to 122x non-timeout, 4,428x with timeout recovery) -- the first demonstration of cross-engine generalization for LLM-based query rewriting. Multi-iteration promotion nearly doubles the win count per round, and 80% of wins are uniquely discovered by a single worker, validating the swarm design. On DSB PostgreSQL, QueryTorque achieves a 60.5% average latency reduction without training, comparable to E^3-Rewrite's 56.4% which requires RL fine-tuning.

Future work. (1) DPO fine-tuning from 500 accumulated preference pairs (win/loss SQL pairs); (2) EXPLAIN-plan-guided worker dispatch; (3) Snowflake and additional engine support; (4) formal equivalence verification via SMT solvers.

References

[zhou2021learned]
X. Zhou, G. Li, C. Chai, and J. Feng.
A learned query rewrite system using Monte Carlo Tree Search.
Proc.\ VLDB Endow., 15(1):46--58, 2021.

[calcite2018]
E. Begoli, J. Camacho-Rodr\'iguez, J. Hyde, M. J. Mior, and D. Lemire.
Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources.

In SIGMOD, pages 221--230, 2018.

[graefe1993volcano]
G. Graefe and W. J. McKenna.
The Volcano optimizer generator: Extensibility and efficient search.
In ICDE, pages 209--218, 1993.

[graefe1995cascades]
G. Graefe.
The Cascades framework for query optimization.
IEEE Data Eng.\ Bull., 18(3):19--29, 1995.

[rbot2024]
Z. Sun, X. Zhou, G. Li, X. Yu, J. Feng, and Y. Zhang.
R-Bot: An LLM-based query rewrite system.
Proc.\ VLDB Endow., 2024.

[e3rewrite2026]
D. Xu, Y. Cui, W. Shi, Q. Ma, H. Guo, J. Li, Y. Zhao, R. Zhang, S. Di,
J. Zhu, K. Zheng, and J. Xu.
E^3-Rewrite: Learning to rewrite SQL for executability,
equivalence, and efficiency.
In AAAI, 2026.

[llmr2_2024]
Z. Li, H. Yuan, H. Wang, G. Cong, and L. Bing.
LLM-R^2$: A large language model enhanced rule-based rewrite
system for boosting query efficiency.
Proc.\ VLDB Endow., 18(1):53--65, 2024.

[quite2025]
Y. Song, H. Yan, J. Lao, Y. Wang, Y. Li, Y. Zhou, J. Wang, and M. Tang.
QUITE: A query rewrite system beyond rules with LLM agents.
CoRR, abs/2506.07675, 2025.

[dsb2021]
B. Ding, S. Chaudhuri, J. Gehrke, and V. R. Narasayya.
DSB: A decision support benchmark for workload-driven and
traditional database systems.
Proc.\ VLDB Endow., 14(13):3376--3388, 2021.

[liu2024lost]
N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and
P. Liang.
Lost in the middle: How language models use long contexts.
ACL, 11:157--173, 2024.

[postgresql]
PostgreSQL Global Development Group.
PostgreSQL: The world's most advanced open source database.

https://www.postgresql.org, 2025.

[wetune2022]
Z. Wang, Z. Zhou, Y. Yang, H. Ding, G. Hu, D. Ding, C. Tang, H. Chen, and J. Li.
WeTune: Automatic discovery and verification of query rewrite rules.
In SIGMOD, pages 94--107, 2022.

[slabcity2023]
R. Dong, J. Liu, Y. Zhu, C. Yan, B. Mozafari, and X. Wang.
SlabCity: Whole-query optimization using program synthesis.
Proc.\ VLDB Endow., 16(11):3151--3164, 2023.

[deepseek_r1]
DeepSeek-AI et al.
DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning.
CoRR, abs/2501.12948, 2025.

[duckdb2019]
M. Raasveldt and H. M\"uhleisen.
DuckDB: An embeddable analytical database.
In SIGMOD, pages 1981--1984, 2019.

[sqlglot]
T. Breddin et al.
SQLGlot: A no-dependency SQL parser, transpiler, optimizer, and engine.
https://github.com/tobymao/sqlglot, 2024.

[openai_o1]
OpenAI.
Learning to reason with LLMs.
OpenAI Blog, September 2024.