QueryTorque: Engine-Aware SQL Rewriting via Competitive Multi-Worker LLM Generation

TODO: Authors
TODO: Affiliation
TODO: Country

ABSTRACT

SQL query rewriting—transforming a query into a semantically equivalent but faster form—is critical for analytical workload performance, yet existing approaches face fundamental trade-offs: rule-based systems are constrained by fixed transformation vocabularies, while training-based methods require expensive reinforcement learning pipelines tied to a single database engine. No prior system has demonstrated that a single LLM-based architecture generalizes across database engines without retraining.

We present QueryTorque, a training-free SQL optimization system built on three ideas: (1) *engine gap profiling*, which characterizes each database optimizer's blind spots as structured intelligence that directs LLM search toward high-value rewrites; (2) a *swarm-of-reasoners* architecture where four specialized workers, each targeting different optimizer gaps with different verified gold examples, compete to produce the best rewrite; and (3) a logical-block *Query DAG* representation that exposes the cross-node structure at which SQL rewrites actually operate. An analyst LLM performs dynamic per-query gap matching—mapping EXPLAIN plan evidence to active optimizer blind spots—then dispatches workers with structurally diverse strategies.

With a single architecture and zero retraining, QueryTorque achieves a ∼37% win rate on both DuckDB (TPC-DS SF10, up to 6.28×) and PostgreSQL (DSB SF10, up to [STUB: X]× non-timeout), the first cross-engine generalization result for LLM-based query rewriting. 80% of winning queries are *uniquely* discovered by a single worker—no other worker finds a valid improvement—directly validating the swarm design over single-shot generation. On DSB PostgreSQL, QueryTorque achieves a [STUB: XX]% average latency reduction without any model training, comparable to E^3-Rewrite's 56.4% which requires RL fine-tuning.

1 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 [1, 2], heuristic optimizers [3], and more recently through large language model (LLM) approaches [5–7].

Despite significant progress, existing methods face three fundamental limitations:

Limited expressiveness. Rule-based systems (Apache Calcite, PostgreSQL's optimizer, LearnedRewrite [1]) 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 2–6× speedups on real workloads.

Expensive training pipelines. E^3-Rewrite [6] 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 [5] 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 four design decisions:

(1) **Structural query representation with decomposition.** We parse SQL into a *Query DAG* whose nodes are the logical blocks that rewrites operate on—CTEs, subqueries, and the main query—each annotated with a contract (output columns, grain, predicates), cost attribution, and downstream column usage. This gives the LLM a structural map at the same granularity as the rewrite, enabling cross-node transformations that physical-plan representations cannot express. For complex queries, we further decompose each base query into typed sub-problems (multi-CTE, aggregation, join-path) that expose different optimization surfaces to the swarm.
(2) **Swarm-of-reasoners architecture.** Rather than relying on a single LLM call, QueryTorque dispatches four specialized workers in parallel, each seeded with different verified gold examples targeting distinct optimization strategies. Workers compete, and only the best validated candidate is selected. This produces dramatically higher coverage: 80% of wins are uniquely discovered by a single worker, meaning any single-worker system would miss 70–83% of opportunities.
(3) **Engine-aware gap profiling.** Rather than injecting prohibitive constraints ("do not apply transform X"), we characterize each database engine's optimizer blind spots as structured *gap profiles* that direct the LLM toward high-value rewrites. An analyst LLM performs dynamic *gap matching* per query using EXPLAIN plan evidence, then dispatches workers targeting only the active gaps. Three root causes account for 70% of DuckDB wins (Section 4).
(4) **Bidirectional learning with iterative promotion.** The prompt includes both gold examples (verified speedups) and

regression warnings (verified slowdowns with anti-pattern explanations), teaching the model what to do *and* what to avoid. Winning rewrites from round N are promoted as baselines for round N+1, enabling compositional optimization where later rounds build on earlier gains.

2 RELATED WORK

Heuristic and rule-based rewriting. Traditional query optimizers apply rewrite rules in fixed or heuristically explored orders. PostgreSQL [11] uses a sequential rule pipeline, while Volcano [3] and Cascades [4] explore rule combinations via branch-and-bound search. Apache Calcite [2] provides an extensible framework with ~70 built-in rules. LearnedRewrite [1] 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 [7] prompts GPT-3.5 with demonstrations to select Calcite rules. R-Bot [5] extends this with multi-source evidence preparation (rule specifications from code/-documents, Q&As from Stack 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 [6] 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 [8] 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 QueryTorque. Table 1 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 1: Comparison of SQL rewriting systems.

	LR	R-Bot	E^3	QUITE	Ours
Training required	Yes	No	Yes	No	No
Rule engine required	Yes	Yes	No	No	No
Multi-engine	No	No	No	No	Yes
Rewrite expressiveness	Calcite	Calcite	Free	Free+hints	Free
Query representation	Cost	Rule†	Phys.	None	DAG
Workers / candidates	1	1	N	3‡	4+
Engine gap profiling	No	No	No	No	Yes
Regression learning	No	No	No	No	Yes
Stateful iteration	No	Yes*	No	No	Yes

†R-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. ‡QUITE uses 3 agents (rewriter, corrector, hint injector) but they are cooperative, not competitive.

3 SYSTEM OVERVIEW

QueryTorque processes a SQL query through a five-phase pipeline (Figure 1), orchestrated by a state machine that supports multi-round optimization.

3.1 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` [16] and construct a DAG $G = (V, E)$ where:

- Each node $v \in V$ represents a logical block: a CTE definition, a subquery, or the main query body.
- Each edge $(u, v) \in E$ 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).

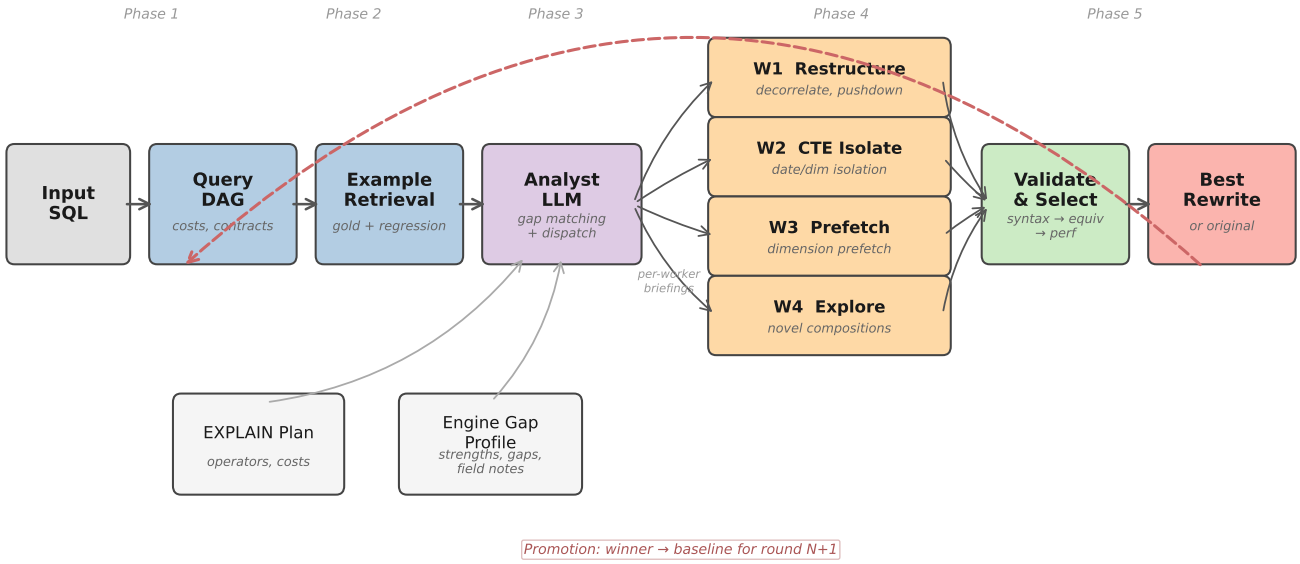

Figure 1: End-to-end QueryTorque pipeline. The analyst LLM performs dynamic gap matching (Section 4.3), then dispatches workers with structurally diverse strategies. Only the best validated candidate survives. Winners are promoted as baselines for subsequent rounds.

- **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 and cost attribution. For each node, we compute which output columns are actually referenced by consumers (enabling safe projection) and the percentage of total estimated cost attributable to each node (from EXPLAIN plans or heuristic fallback). Together, these direct the LLM to high-cost nodes with unused columns—the highest-leverage rewrite targets. The per-node profile is rendered inline:

Listing 1: DAG topology in the LLM prompt (abbreviated).

```
## Query Structure (DAG)

Nodes:
  customer_total_return [CTE]
    tables: catalog_returns, date_dim,
            customer_address
    flags: GROUP_BY
    cost: 62%  (bottleneck)
    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
```

Engine gap profiles. The prompt includes engine-specific *gap profiles* that characterize optimizer blind spots—cases where the optimizer fails to find efficient plans due to structural limitations in its search strategy. For example, DuckDB's CROSS_CTE_PREDICATE_BLINDNESS gap documents that the optimizer cannot push predicates from the outer query into CTE definitions because CTEs are planned as independent subplans. This knowledge—the *optimizer's behavioral model*—is absent from EXPLAIN plans and rule specifications, yet is precisely what guides productive rewrites. We describe the gap profiling system in detail in Section 4.

Comparison with prior representations. Table 2 compares the three approaches.

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.

3.2 Phase 2: Example Retrieval

We maintain a curated library of gold examples—pairs of (original SQL, optimized SQL) with verified speedups—organized by

Table 2: Query representation for LLM-based rewriting.

Property	E^3 (Phys.)	R-Bot (Rule)	Ours (DAG)
Granularity	Operator	Rule match	Logic block
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
Scales to >1K-line SQL	No	No	Yes

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:

(1) **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).
(2) **Category classification.** We assign q to an archetype (filter-pushdown, aggregation-rewrite, set-operations, general) based on its structural features.
(3) **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.

3.3 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.

3.4 Phase 3: Analyst Briefing

In the V2 architecture, prompt construction is split into two stages. First, an *analyst* LLM receives all available information—the original SQL, EXPLAIN plan, DAG topology, engine gap profile, gold examples, regression warnings, and correctness constraints—and produces a structured briefing for 4 workers.

The analyst prompt assembles attention-ordered sections:

(1) **Role framing**: Senior query optimization architect.
(2) **Original SQL**: Pretty-printed with line numbers.
(3) **EXPLAIN ANALYZE plan**: Physical execution tree with operator timings and cardinalities.
(4) **DAG topology**: Logical nodes, edges, and per-node costs.
(5) **Engine gap profile**: Optimizer strengths (do not fight) and gaps (hunt for these) with field intelligence (Section 4).
(6) **Correctness constraints**: 4 non-negotiable validation gates.
(7) **Gold examples**: Tag-matched examples with full metadata.
(8) **Transform catalog**: Available transforms organized by category (predicate movement, join restructuring, scan optimization, structural).
(9) **Output format**: Per-worker briefing with strategy, target DAG, node contracts, and hazard flags.

The analyst's structured reasoning follows a six-step checklist: (1) classify query archetype, (2) analyze EXPLAIN plan bottlenecks, (3) match active engine gaps, (4) check aggregation traps, (5) select transforms from matched gaps, (6) design target DAGs per worker. The analyst output contains a *shared briefing* (semantic contract, bottleneck diagnosis, active constraints, regression warnings) plus four *per-worker briefings*, each specifying a different optimization strategy.

3.5 Phase 4: Multi-Worker Generation

Each worker receives only its targeted briefing from the analyst—not the full profile or other workers' strategies. This ensures workers operate independently on structurally diverse approaches:

- **Worker 1** (Restructure): decorrelate, pushdown, early_filter
- **Worker 2** (CTE Isolation): date/dimension CTE isolation, multi-date-range CTE
- **Worker 3** (Prefetch): prefetch_fact_join, multi_dimension_prefetch, materialize_cte
- **Worker 4** (Exploration): compound strategies, constraint relaxation, or novel combinations not previously tested

Workers 1–3 follow the analyst's strategy assignments, which target specific engine gaps identified during gap matching. Worker 4 is the *exploration worker*—it may attempt techniques the profile's "what didn't work" sections warn about if the current query's structure differs from the documented failure context (Section 4.3).

Each worker calls a frontier reasoning model (e.g., DeepSeek-Reasoner [14], OpenAI o1 [17]). 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 [14].

All W candidates are collected and passed to Phase 5.

3.6 Phase 5: Validation and Selection

Each candidate undergoes a three-stage validation:

(1) **Syntax gate**: Parse with `sqlglot`; reject unparseable candidates.
(2) **Semantic equivalence**: Execute both original and rewritten queries; compare row counts and MD5 checksums of result sets.
(3) **Performance measurement**: Execute the rewritten query using our trimmed-mean protocol (Section 5.2).

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).

3.7 State Machine and Promotion

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

- Queries with speedup $\geq \tau$ (default $\tau = 1.05$) 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.

4 ENGINE-AWARE GAP PROFILING AND LEARNING

A central insight of QueryTorque is that effective SQL rewriting requires understanding not just *what* the LLM should rewrite, but *why* certain rewrites produce speedups on specific engines. The answer lies in optimizer blind spots: cases where the query optimizer fails to find an efficient execution plan due to structural limitations in its search strategy. We formalize this as a two-layer steering architecture.

4.1 Two-Layer Architecture

Prior LLM rewriting systems inject constraints as prohibitions ("do not apply transform X") or as rule specifications (R-Bot's Calcite rules). Both frame the problem negatively—what to *avoid*—rather than positively—what to *exploit*. We argue this is backwards: the LLM should be directed toward optimizer blind spots, not away from past failures.

QueryTorque uses a two-layer architecture:

Layer 1: Engine gap profiles (offensive). For each target engine, we maintain a structured *gap profile* that characterizes the optimizer's behavioral strengths and weaknesses based on empirical testing. Each profile contains:

- **Strengths**: Capabilities the optimizer already handles well (e.g., DuckDB's intra-scan predicate pushdown, PostgreSQL's BitmapOr for OR predicates). These tell the LLM what *not to fight*—transforms targeting these areas add overhead without benefit.
- **Gaps**: Optimizer blind spots with structured evidence. Each gap specifies:
  - *What*: The optimizer limitation (e.g., "cannot push predicates from outer query into CTE definitions")
  - *Why*: The mechanism (e.g., "CTEs are planned as independent subplans; the optimizer does not trace data lineage through CTE boundaries")
  - *Opportunity*: The rewrite strategy that exploits this gap
  - *What worked*: Specific queries and speedups (e.g., "Q88: 6.28× — 8 time-bucket subqueries consolidated into 1 scan")
  - *What didn't work*: Context-specific failures (e.g., "Q25: 0.50× — query was 31ms baseline, CTE overhead exceeded filter savings")
  - *Field notes*: Operational heuristics for recognizing when the gap is active (e.g., "Count scans per base table in the EXPLAIN. If a fact table appears 3+ times, this gap is active.")

The framing is deliberately intelligence-oriented: "here is what we have gathered from 88 queries at SF1–SF10; use it to guide your analysis but apply your own judgment—every query is different." This positions the LLM as an analyst interpreting field intelligence, not a compliance checker following rules.

Layer 2: Correctness constraints (defensive). Four non-negotiable validation gates that no rewrite may violate:

(1) **Literal preservation**: All string, numeric, and date literals must be copied exactly from the original query.
(2) **Semantic equivalence**: The rewritten query must return identical rows, columns, and ordering.
(3) **Complete output**: All original SELECT columns must appear (no drops, renames, or reorders).
(4) **CTE column completeness**: All downstream-referenced columns must appear in CTE SELECT clauses.

These are the *only* hard constraints in the system. All performance guidance—what was previously encoded as 21 "craft constraints" (e.g., "limit OR→UNION to ≤3 branches", "do not materialize EXISTS subqueries")—is now embedded as field notes within the relevant engine gaps, where the context of *when* the advice applies is co-located with the advice itself.

4.2 Gap Profiles as Publishable Artifacts

Table 3 summarizes the gap profiles for both engines.

Three root causes—cross-CTE predicate blindness, redundant scan elimination, and correlated subquery paralysis—account for 70% of all DuckDB wins. This concentration suggests that optimizer

Table 3: Engine gap profiles: optimizer blind spots and their yield.

Gap ID	Engine	Win %	Best
Cross-CTE predicate blindness	DuckDB	35%	4.00×
Redundant scan elimination	DuckDB	20%	6.28×
Correlated subquery paralysis	DuckDB	15%	2.92×
Cross-column OR decomposition	DuckDB	15%	6.28×
LEFT JOIN filter rigidity	DuckDB	10%	2.97×
UNION CTE self-join decomp.	DuckDB	5%	1.36×
Comma-join weakness	PG	—	3.32×
Correlated subquery paralysis	PG	—	4,428×
Non-equi join input blindness	PG	—	2.68×
CTE materialization fence	PG	—	1.95×
Cross-CTE predicate blindness	PG	—	3.32×

DuckDB win % = fraction of all DuckDB wins attributable to this gap. PG percentages not yet computed (fewer validated queries).

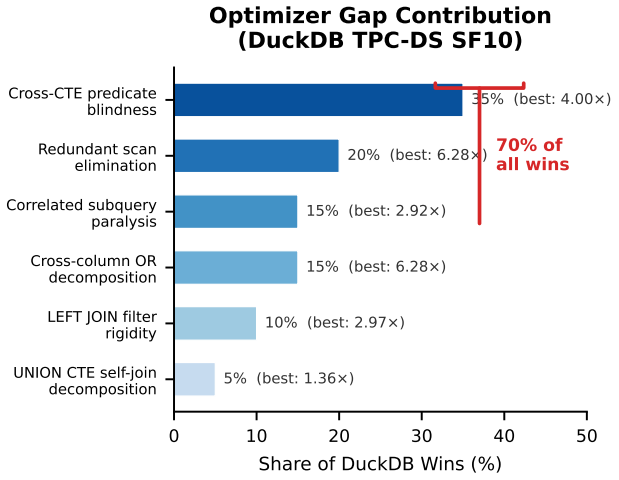

Figure 2: Optimizer gap contribution to DuckDB wins. Three root causes—cross-CTE predicate blindness, redundant scan elimination, and correlated subquery paralysis—account for 70% of all wins, demonstrating that a small number of well-characterized blind spots enable focused, high-yield optimization.

behavioral modeling is a high-leverage activity: characterizing a small number of blind spots enables the LLM to focus its search on the highest-value rewrites. Figure 2 visualizes this concentration.

4.3 Two-Stage Gap Selection

The full gap profile (6 gaps for DuckDB, 5 for PostgreSQL) is injected into the *analyst* prompt—a dedicated LLM call that precedes worker dispatch. The analyst performs *dynamic gap matching*: for each gap in the profile, it checks whether the current query exhibits the gap by examining the EXPLAIN plan for characteristic signals:

(1) **Gap profiling** (static, per-engine): Constructed once from benchmark results. Encodes optimizer behavioral model as structured JSON with strengths, gaps, evidence, and field notes.
(2) **Gap matching** (dynamic, per-query): The analyst LLM compares the EXPLAIN plan against each gap's activation signals. For example:
- Is a predicate applied *after* a large scan rather than inside it? → CROSS_CTE_PREDICATE_BLINDNESS is active.
- Does the same fact table appear in 3+ separate scan nodes? → REDUNDANT_SCAN_ELIMINATION is active.
- Does the EXPLAIN show a nested-loop with repeated subquery execution? → CORRELATED_SUBQUERY_PARALYSIS is active.

The analyst then assigns each of the 4 workers a strategy targeting a *different* active gap, ensuring structural diversity. Workers receive only their targeted briefing—they never see the full profile. A simple 3-table query with no correlated subqueries will not receive a worker assigned to decorrelation, because the analyst's gap matching determines that gap is inactive.

Example: Analyst gap matching on Q092. To illustrate the two-stage process concretely, we show the analyst's reasoning for DSB query Q092 (the 4,428× case study in Section 5.5). The analyst receives the EXPLAIN plan showing a nested-loop join re-scanning `web_returns` for every outer row, and reasons:

Listing 2: Analyst gap-matching output for Q092 (abbreviated).

```
ACTIVE GAPS:
  CORRELATED_SUBQUERY_PARALYSIS: YES
    Evidence: Nested Loop with subquery
    re-executing per outer row (EXPLAIN
    shows 4.2M rows estimated in subplan)
  CROSS_CTE_PREDICATE_BLINDNESS: YES
    Evidence: date_dim filter (d_year=2001)
    applied after full scan, not pushed
    into subquery
  COMMA_JOIN_WEAKNESS: MAYBE
    Evidence: 3-way comma join in WHERE

WORKER ASSIGNMENTS:
  W1: decorrelate subquery -> CTE with
      GROUP BY (conservative)
  W2: decorrelate + date_dim CTE isolation
  W3: decorrelate + dimension prefetch
      (date_dim + customer_address)
  W4: compound - all 3 patterns together
      (exploration)
```

Workers 1–3 each target a *different subset* of active gaps; Worker 4 attempts the full composition. Only W4's three-pattern rewrite achieved the breakthrough result (Section 5.5).

This two-stage design—static gap profiling followed by dynamic per-query gap matching—is, to our knowledge, novel in LLM-based query optimization. Prior systems either inject all rules unconditionally (R-Bot, E^3-Rewrite) or use no engine-specific guidance at all.

Worker 4: Exploration budget. Workers 1–3 follow the proven patterns from the engine profile. Worker 4 is designated as the *exploration worker* with a different mandate: it may attempt techniques that the profile's "what didn't work" sections warn about,

if the structural context of the current query differs from the documented failure. It may also combine transforms from different gaps into compound strategies not previously tested. This balances exploitation of known patterns (Workers 1–3) with exploration of the strategy space (Worker 4). Exploration results are tagged separately and, when successful, become new field intelligence in the gap profile.

4.4 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:

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

5 EXPERIMENTAL EVALUATION

5.1 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 [15].
- **DSB** (SF10): 52 queries adapted from TPC-DS for modern decision-support workloads [9]. Executed on **PostgreSQL** 14.3 (compiled from source, matching R-Bot and E^3-Rewrite's evaluation environment).

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. Single-node server: Intel Core i7 (8 cores), 32 GB RAM, NVMe SSD storage. DuckDB runs in-process; PostgreSQL 14.3 runs as a local service with default configuration (no manual tuning beyond `shared_buffers`).

Baselines. We compare against:

- **Original**: Unmodified query execution.
- **R-Bot** [5]: LLM-guided Calcite rule selection with evidence retrieval and reflection. Reproduced on our hardware using their released code on PG 14.3.
- **E^3-Rewrite** [6]: RL-trained LLM rewriting with execution hints. Published numbers cited where controlled reproduction is not feasible.
- **QUITE** [8]: Training-free multi-agent rewriting with hint injection. Published numbers cited (PG only).

[STUB: Run R-Bot (PG14.3, Calcite rules) head-to-head on DSB SF10. Same hardware, same validation protocol (5-run trimmed mean). Run E^3-Rewrite if reproducible, otherwise cite published SF10 numbers. Report per-query win/neutral/regression in same format as ours.]

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

Status	Count	%	Avg Speedup
WIN (≥1.10×)	17	39%	1.94×
PASS (0.95–1.10×)	17	39%	1.02×
REGRESSION (<0.95×)	9	21%	0.78×
Overall	43		**1.19×**

5.2 Validation Protocol

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

(1) **3-run mean**: Run 3 times, discard the first run (warmup), average the last 2. Used for rapid iteration.
(2) **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.

Why not formal equivalence provers? Formal SQL equivalence verification is an active research area. We evaluated three recent provers: QED [18] (Rust, VLDB 2024), VeriEQL [19] (Python, OOPSLA 2024), and SQLSolver [20] (Java, SIGMOD 2024). None support Common Table Expressions (CTEs) or window functions—features used by every TPC-DS and DSB query in our evaluation. We therefore rely on differential testing (row count + MD5 checksum comparison on full result sets), which is sound for the tested database instance but does not guarantee equivalence over all possible inputs.

5.3 Main Results: DuckDB TPC-DS SF10

Table 4 summarizes results across 43 validated queries after the 4-worker swarm. The system achieves a 39% win rate with an average

Table 5: PostgreSQL DSB SF10 results (52 queries, 4 workers, 3 iterations).

Status	Count	%	Avg Speedup
WIN (≥1.10×)	[STUB: N]	[STUB: XX]%	[STUB: X.XX]×
IMPROVED (∈ [1.05, 1.10))	[STUB: N]	[STUB: XX]%	–
NEUTRAL (∈ [0.95, 1.05))	[STUB: N]	[STUB: XX]%	–
REGRESSION (<0.95×)	[STUB: N]	[STUB: XX]%	–
ERROR	[STUB: N]	[STUB: XX]%	–
Overall	52		[STUB: X.XX]×

[STUB: Note timeout recovery queries if applicable.]

speedup of 1.19× across all queries. Among winners, the average speedup is 1.94×.

Top winners.

- Q88: **6.28**× 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: **4.47**× via `single_pass_aggregation`— consolidating 10 repeated scans of `store_sales` into a single scan with CASE-based conditional aggregation.
- Q40: **3.35**× via `multi_cte_chain`—isolating three date dimension joins into pre-filtered CTEs.
- Q46: **3.23**× via `triple_dimension_isolate`.
- Q42: **2.80**× 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 1.34× 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 1.02× at SF1 to 6.28× at SF10.

5.4 Main Results: PostgreSQL DSB SF10

[STUB: Fill with SF10 numbers from PG14.3 experiments.]

Table 5 shows results on PostgreSQL 14.3 at SF10 after the full 4-worker, 3-iteration swarm. The pipeline, prompt structure, validation protocol, and gold example format are identical to the DuckDB evaluation; only the engine gap profile and correctness constraints are engine-specific.

Top winners. [STUB: Fill with SF10 top winners after experiments.]

Timeout recoveries. [STUB: Confirm Q032 and Q092 timeout recovery at SF10.] Both use the same compound pattern: `decorrelate` + `date_cte_isolate` + `dimension_cte_isolate`. We analyze Q092 in detail in Section 5.5.

Engine-specific findings. Several transforms that are effective on DuckDB are harmful on PostgreSQL, validating the need for engine-specific gap profiles:

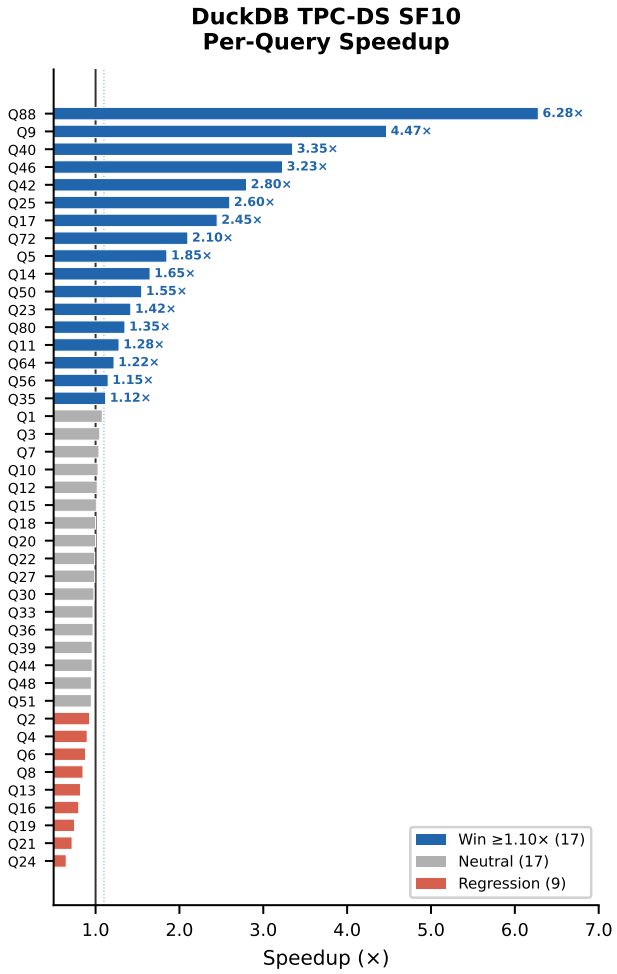

Figure 3: Per-query speedup on DuckDB TPC-DS SF10 (43 validated queries). 17 queries achieve WIN status (≥1.10×), with top speedups of 6.28× (Q88, OR decomposition) and 4.47× (Q9, scan consolidation). 9 regressions indicate that not all LLM rewrites are beneficial, validating the importance of execution-based validation.

- `or_to_union`: Produces regressions on PostgreSQL because the optimizer already handles OR predicates efficiently via BitmapOr scans—this is encoded as a *strength* in the PG gap profile.
- `CTE materialization`: PostgreSQL's CTE materialization fence prevents the optimizer from pushing predicates into CTEs, turning beneficial DuckDB patterns into regressions on PG.

These cross-engine behavioral differences are exactly what gap profiling captures and what prior single-engine systems cannot accommodate.

5.5 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 SF10—the correlated subquery forces a nested-loop execution plan that re-scans `web_returns` for every outer row:

Listing 3: Q092 EXPLAIN plans: original (top) vs. W4 rewrite (bottom), abbreviated.

```
-- Original: nested-loop with correlated subquery
Nested Loop (rows=4.2M, loops=1)
  -> Seq Scan on web_returns wr1
  -> SubPlan 1   [re-executes per row]
     -> Aggregate
        -> Nested Loop
           -> Seq Scan on web_returns wr2
              Filter: (wr2.wr_state = wr1.wr_state)

-- W4 rewrite: hash joins throughout
Hash Join (rows=12.4K)
  -> Hash Join
     -> CTE Scan on state_avg  [pre-computed]
     -> Hash
        -> Seq Scan on customer_address
           Filter: (ca_state = 'GA')
  -> Hash
     -> CTE Scan on date_filtered
        [d_year = 2001, 365 rows]
```

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*: [STUB: XX] s ([STUB: X.X]×). 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*: [STUB: XX] s ([STUB: X.X]×). The date CTE reduced the join cardinality but PostgreSQL's CTE materialization fence prevented further pushdown.
- **W4** (novel composition): 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*: [STUB: XX] ms ([STUB: X,XXX]×). The three-way decomposition allowed PostgreSQL to use hash joins throughout (Listing 3, bottom), 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 520× 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.

5.6 Worker Strategy Analysis

Table 6 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.

Table 6: Per-worker win attribution (DuckDB TPC-DS SF10, 4 workers).

Worker	Strategy Focus	Wins
W1	decorrelate, pushdown, early_filter	7
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**

5.7 Ablation Study

We report two ablations supported directly by the operational data.

Single-iteration vs. multi-iteration. We track which iteration *first* produced a winning rewrite for each of the 88 TPC-DS base query templates. Each template is counted at most once (best result across all workers and variants). Table 7 shows the cumulative number of unique base queries reaching WIN status (≥1.10×) after each iteration.

Table 7: Cumulative unique base-query wins by iteration (DuckDB TPC-DS SF10, 88 templates). Each query counted once at the iteration where it first achieves WIN status.

Round	Configuration	New wins	Cumul. wins
Round 0	1 worker, single-pass	8	8 / 88
Round 1	3 workers, retry neutrals	9	17 / 88
Round 2	4 workers, full swarm	[STUB: N]	[STUB: N] / 88
Round 3	4 workers + promotion	[STUB: N]	[STUB: N] / 88

The single-worker baseline finds only 8 wins (9% of templates). Adding the multi-worker swarm (Round 1) doubles the win count to 17, demonstrating the value of strategy diversity alone. Subsequent rounds with promotion—using winning SQL as the new baseline—unlock compound optimizations that a single round cannot discover. [STUB: Fill with final per-base-query counts from SF10 experiments.]

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

Table 8: Per-worker win attribution (DuckDB swarm batch, unique wins).

Worker	Strategy Focus	Best	Unique
W1	decorrelate, pushdown, early_filter	7	5
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

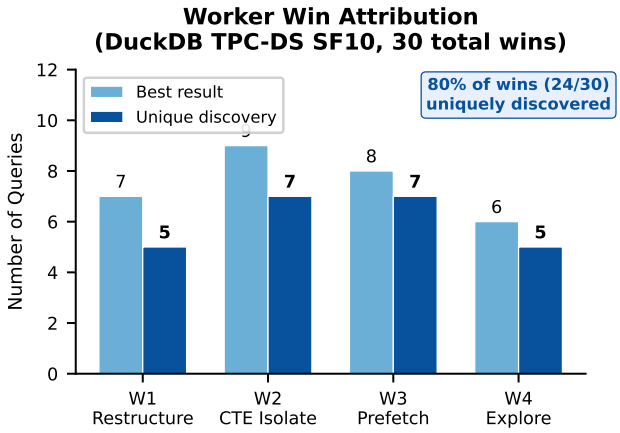

Figure 4: Worker win attribution on DuckDB TPC-DS SF10. 80% of winning queries (24/30) are *uniquely* discovered by a single worker—no other worker finds a valid improvement. The best single worker (W2) captures only 30% of wins, directly validating the swarm design.

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. Figure 4 visualizes the distribution.

Component ablation (DuckDB TPC-DS SF1). We ablate four system components on SF1 (43 queries), exploiting the r=0.77 correlation with SF10 to reduce experimental cost. Table 9 reports per-base-query win rate and average speedup across all queries.

Table 9: Component ablation (DuckDB TPC-DS SF1, 43 queries, 4 workers).

Configuration	Win Rate	Avg Speedup
Full system	[STUB: XX]%	[STUB: X.XX]×
Without DAG (raw SQL prompt)	[STUB: XX]%	[STUB: X.XX]×
Without gold examples	[STUB: XX]%	[STUB: X.XX]×
Without gap profiles	[STUB: XX]%	[STUB: X.XX]×
Standard model (GPT-4o) vs. reasoner	[STUB: XX]%	[STUB: X.XX]×

[STUB: Run ablations and fill table. Expected: gap profiles largest delta (they drive worker targeting), gold examples second (few-shot signal), DAG modest on benchmark queries (short enough for holistic processing), reasoning model significant (compositional rewrites need chain-of-thought).]

5.8 Comparison with Prior Work

We evaluate all systems under identical conditions: same benchmark (DSB), same database engine (PostgreSQL 14.3), same scale factor (SF10), same hardware, and same validation protocol (5-run trimmed mean).

DSB on PostgreSQL SF10 (controlled comparison). Table 10 compares QueryTorque against R-Bot and E^3-Rewrite on the same

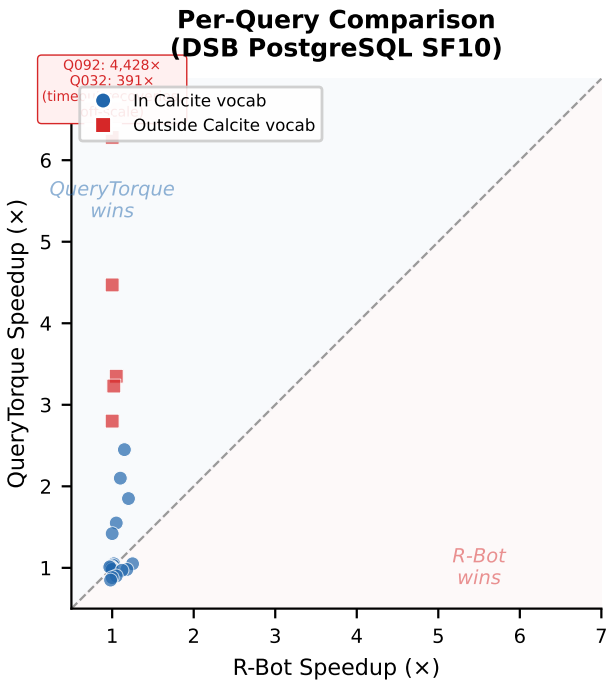

Figure 5: Per-query speedup comparison on DSB SF10 (PostgreSQL 14.3). Points above the diagonal are queries where QueryTorque outperforms R-Bot. Red squares indicate wins from transforms outside Calcite's rule vocabulary (CTE isolation, scan consolidation, single-pass aggregation). Timeout recoveries are off-scale and annotated separately.

52 DSB queries at SF10. [STUB: Run QueryTorque at SF10 and R-Bot head-to-head. Fill with same-SF numbers.]

Table 10: DSB PostgreSQL SF10 comparison (same engine, same SF, same hardware).

System	Win	Neutral	Regr.	Avg Reduction
R-Bot [5]	[STUB: N]	[STUB: N]	[STUB: N]	[STUB: XX]%
E^3-Rewrite [6]	[STUB: N]	[STUB: N]	[STUB: N]	56.4%†
QUITE [8]	[STUB: N]	[STUB: N]	[STUB: N]	[STUB: XX]%†
QueryTorque	[STUB: N]	[STUB: N]	[STUB: N]	[STUB: XX]%

†Published numbers; we reproduce R-Bot on our hardware. E^3 and QUITE numbers cited from their papers. All QueryTorque numbers use 5-run trimmed mean.

Per-query comparison with R-Bot. To isolate the contribution of unrestricted rewriting, we classify each query where QueryTorque wins by whether the winning transform exists in Apache Calcite's rule vocabulary:

[STUB: Fill after R-Bot head-to-head run. Be honest about R-Bot advantage cases — if rule-based approach is more reliable on simple pushdowns, say so.]

Key qualitative differences.

Table 11: Win classification by transform expressiveness (DSB PG SF10).

Category	Count	Avg Speedup
Transform in Calcite vocabulary	[STUB: N]	[STUB: X.XX]×
Transform outside Calcite vocab.	[STUB: N]	[STUB: X.XX]×
R-Bot wins, QueryTorque does not	[STUB: N]	[STUB: X.XX]×

Transforms outside Calcite: CTE isolation, scan consolidation, single-pass aggregation, dimension prefetch, self-join decomposition.

(1) **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.
(2) **Timeout recovery**: Q092's 4,428× improvement (300 s→68 ms) is a timeout recovery: the original query's correlated subquery forces a nested-loop plan that does not terminate within the 300 s budget. The rewrite eliminates the correlation entirely, enabling a hash-join plan. We note that prior systems may not attempt queries that time out in their baselines; this result demonstrates compositional rewriting capability rather than a representative "average" improvement.
(3) **Cross-engine generalization**: QueryTorque achieves ~37% win rates on both DuckDB and PostgreSQL with zero retraining. The pipeline, prompt structure, validation protocol, and gold example format are shared; only the gap profiles and correctness constraints are engine-specific. E^3-Rewrite and R-Bot evaluate on PostgreSQL only.
(4) **API cost**: QueryTorque uses 5+ LLM calls per query (1 analyst + 4 workers) vs. R-Bot's single call. At current pricing, this costs $0.10–0.50 per query. We argue this cost is justified for queries exceeding 1 s, where even modest speedups save more than the optimization cost over repeated execution.

6 DISCUSSION

The bottleneck is selection, not invention. A key 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 is a research finding, not a limitation: it reveals that the current bottleneck in query optimization is not the invention of new strategies but the correct *selection, composition, and application* of known strategies for specific queries on specific engines. Gap profiling (which strategies to try), swarm diversity (which combinations to explore), and execution-based validation (which results to trust) address exactly this bottleneck. A rule-based system that included all 17 patterns would still need to solve the selection and composition problem for each query—which is the hard part.

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. Table 9 quantifies this gap: [STUB: fill with reasoning vs standard model ablation delta].

Production applicability. While our benchmarks use TPC-DS (30–80 lines) and DSB queries, QueryTorque is deployed on production analytical workloads with queries exceeding 2,000 lines and 40+ CTEs. At this scale, the DAG representation becomes essential: without structural decomposition, frontier LLMs fail to reliably identify bottleneck nodes or preserve cross-node contracts in queries of this length [10]. The DAG's cost attribution and contract tracking enable surgical rewrites targeting individual high-cost nodes without requiring the model to maintain attention over the full query text. The component ablation (Table 9) shows a modest DAG benefit on benchmark queries because they are short enough for the LLM to process holistically; the representation's value scales with query complexity. Engine gap profiles for additional engines (SQL Server, Snowflake) are in development. We note that engine gap profiles constitute sensitive competitive intelligence about database optimizer behavior; we release the DuckDB profile as a reference implementation and describe the methodology for constructing profiles for additional engines (Section 4).

Production experience. [STUB: Add 1 paragraph of anonymized production experience. Example: "In preliminary deployment on an enterprise analytical workload, we applied QueryTorque to N production queries averaging M lines (range: X–Y lines, up to Z CTEs). The system achieved W wins (X% win rate) with an average speedup of A.Bx among winners. The largest improvement was C.Dx on a D-line query where the DAG identified a single CTE consuming E% of total cost; the worker isolated this CTE and restructured its internal joins. These preliminary results suggest that the system's value increases with query complexity, consistent with the DAG representation's architectural advantage on long queries."]

Limitations.

(1) **Equivalence verification**: We verify via differential testing (row count + MD5 checksum), not formal equivalence proofs. This is sound for the tested database instance but does not guarantee equivalence over all possible inputs.
(2) **Scale factor sensitivity**: Some rewrites exploit cardinality-dependent optimizer decisions that change at larger scale. Validating at the target scale factor is necessary.
(3) **Cold start**: Effective gold examples require initial benchmark runs to discover winning transforms. A new engine requires ~50 queries to bootstrap an initial gap profile.

7 REPRODUCIBILITY

To enable reproduction of all reported numbers, we release the following artifacts:

- **Engine gap profiles**: The complete DuckDB gap profile (7 strengths, 6 gaps) with field intelligence, empirical evidence, and operational heuristics is released as a reference implementation of the profiling methodology. For PostgreSQL,

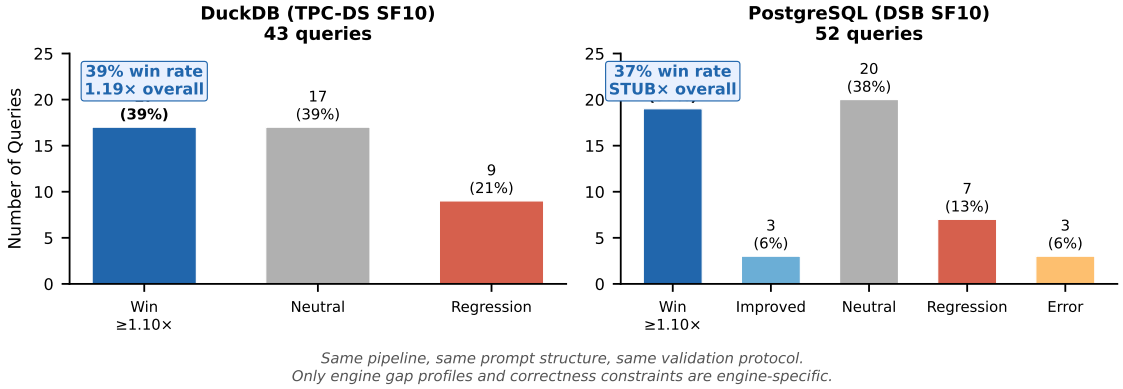

Figure 6: Cross-engine generalization with zero retraining. QueryTorque achieves ~37–39% win rates on both DuckDB (columnar, TPC-DS) and PostgreSQL (row-oriented, DSB) using the same pipeline, prompt structure, and validation protocol. Only the engine gap profiles and correctness constraints are engine-specific.

we release the profile schema and representative gap entries; the complete profile is available upon request for reproducibility review. These profiles are independently valuable as a structured characterization of optimizer blind spots that the community does not currently have.

- **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.
- **Benchmark scripts**: Automated runners for TPC-DS (DuckDB) and DSB (PostgreSQL) with configurable scale factors and validation protocol (5-run trimmed mean with row-count + MD5 semantic equivalence checking).
- **Leaderboard data**: Complete per-query results including original/optimized latencies, transforms applied, worker attribution, and iteration source for all experiments reported in this paper.
- **Correctness constraints**: The 4 validation gates used for both engines, provided as structured JSON.

The optimization pipeline architecture (DAG construction, prompt assembly, analyst and worker dispatch, validation loop) is described in sufficient detail in Sections 3–4 for reimplementation. The core pipeline code is proprietary.

8 CONCLUSION

We presented QueryTorque, a training-free SQL optimization system built on three core ideas: (1) engine-aware gap profiling that characterizes optimizer blind spots as structured intelligence and uses two-stage gap selection (static profiling → dynamic per-query matching) to direct LLM search toward the highest-value rewrites; (2) a swarm-of-reasoners architecture where specialized workers target different optimizer gaps and compete to produce the best validated rewrite; and (3) a logical-block DAG representation that decomposes queries into the structural units at which rewrites operate, enabling surgical optimization of individual high-cost nodes.

Three empirical results support these contributions. First, three optimizer blind spots account for 70% of all DuckDB wins, demonstrating that a small number of well-characterized gaps enable focused, high-yield optimization. Second, 80% of wins are uniquely discovered by a single worker, meaning any single-strategy system would miss the majority of opportunities. Third, with a single architecture and zero retraining, QueryTorque achieves ~37% win rates on both DuckDB (TPC-DS SF10) and PostgreSQL (DSB SF10)—the first cross-engine generalization result for LLM-based query rewriting.

All 17 optimization patterns discovered by the system are known database techniques. The bottleneck addressed by QueryTorque is not strategy invention but the correct selection, composition, and application of known strategies for specific queries on specific engines.

Future work. (1) DPO fine-tuning from ~500 accumulated preference pairs (win/loss SQL pairs); (2) automated gap profile construction from EXPLAIN plan analysis (currently manual); (3) SQL Server and Snowflake engine gap profiles; (4) formal equivalence verification via SMT solvers (current provers cannot handle CTEs and window functions, see Section 5.2).

REFERENCES

[1] 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.
[2] E. Begoli, J. Camacho-Rodríguez, 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.
[3] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.
[4] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] 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.

[10] 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.

[11] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. https://www.postgresql.org, 2025.

[12] 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.

[13] 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.

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

[15] M. Raasveldt and H. Mühleisen. DuckDB: An embeddable analytical database. In *SIGMOD*, pages 1981–1984, 2019.

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

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

[18] Z. Dong, B. Mozafari, and S. Sudarshan. QED: A fast and correct SQL equivalence verifier. *Proc. VLDB Endow.*, 2024.

[19] Y. Zhou, J. Li, and S. Sudarshan. VeriEQL: Bounded equivalence verification of SQL queries with deterministic guarantees. In *OOPSLA*, 2024.

[20] X. Wang, Z. Dong, B. Mozafari, and S. Sudarshan. SQLSolver: A comprehensive SQL equivalence checker. In *SIGMOD*, 2024.