DuckDB optimizer gaps: an adversarial analysis for Query Torque

DuckDB's query planner has systematic blind spots that an external SQL optimizer can exploit. Despite implementing state-of-the-art algorithms like DPccp join ordering and vectorized execution, the planner exhibits predictable failures in cardinality estimation, predicate pushdown through blocking operators, and algebraic rewrites that an LLM-powered tool can leverage for **10-1000x performance improvements** on specific query patterns. This analysis maps the attack surface.

Join ordering and cost model failures create the largest optimization window

DuckDB's join order optimizer uses dynamic programming (DPccp/DPhyp), but its **cost model has a critical flaw**: it calculates join cost as simply the maximum cardinality of left/right relations, ignoring accumulated costs from previous joins. GitHub This causes catastrophic plan selection on complex queries.

The most severe documented case is **GitHub Issue #3525**, where TPC-DS Query 37 at SF1000 took **1.5 hours** instead of 5 minutes because the optimizer placed large tables at the bottom of the join tree, creating massive intermediate results. GitHub The cost calculation at `join_order_optimizer.cpp:214` treats all join orderings as equivalent cost when dominated by a single large table. GitHub github

Additional join weaknesses include:

- **Greedy fallback on complex join graphs** — when search space exceeds internal thresholds, DuckDB switches to heuristics that produce suboptimal plans
- **No "interesting orderings"** — unlike System R, DuckDB doesn't propagate sort orders for downstream ORDER BY/GROUP BY optimization
- **Semi/anti-join inequality gaps** — correlated subqueries with inequality conditions (e.g., `l_suppkey <> outer.l_suppkey`) historically triggered INNER JOIN instead of proper semi-joins, causing exponential blowup on TPC-H Q21 GitHub

Query Torque opportunity: Implement Selinger-style cost modeling that accumulates intermediate cardinalities. Reorder joins to place highly selective predicates and smaller intermediates earlier. Transform inequality semi-joins to hash-based patterns.

Cardinality estimation runs nearly blind on real workloads

DuckDB maintains **minimal statistics** compared to traditional databases — no histograms, no samples, no multi-column statistics, no heavy-hitter tracking. Per the official MSc thesis on DuckDB join ordering:

> "The intended workflow for DuckDB does not allow it to collect similar statistics as many sessions are short-lived, or data is queried directly from Parquet or CSV files."

Statistic Type	DuckDB Native	Parquet Files
Min/Max zonemap	✓	✓
HyperLogLog distinct count	✓	✗
Histograms	✗	✗
Multi-column correlation	✗	✗
Samples	✗	✗

This creates predictable estimation failures:

- **Correlation blindness** — `WHERE city='NYC' AND state='NY'` assumes independence, massively overestimating selectivity
- **Skew handling** — uniform distribution assumptions fail on Zipfian data
- **Parquet cardinality** — official docs warn that "Parquet files do not have the hyperloglog statistics that improve accuracy of cardinality estimates"
- **Hive partition filtering** — PR #18612 documents that cardinality estimates aren't updated after Hive filter pushdown, causing wrong hash table build sides

Query Torque opportunity: Build runtime histograms, detect column correlations through sampling, inject cardinality hints through query restructuring.

Predicate pushdown fails at blocking operator boundaries

DuckDB's filter pushdown hits hard stops at several operator types:

Window functions completely block pushdown (Issue #10352). Even when filtering on the PARTITION BY column, predicates won't push through:

```sql
-- Filter CANNOT push through to table scan despite matching partition key
SELECT *, LAG(sessionid) OVER (PARTITION BY partition_year_month ORDER BY ts)
FROM sessions
WHERE partition_year_month = '2024-01'
```

UNNEST/LATERAL joins block pushdown (Issue #13861, #18653). A query filtering after CROSS JOIN UNNEST processes all rows before filtering: github

```sql
```

```sql
-- Takes 3.45s: cross-joins 30M rows, then filters
SELECT id, value FROM test_table
CROSS JOIN unnest(values) AS values(value)
WHERE id = 871000;

-- Takes milliseconds: subquery pre-filters
SELECT id, value FROM (
  SELECT * FROM test_table WHERE id = 871000
) t CROSS JOIN unnest(values) AS values(value);
```

Materialized CTEs block pushdown by design. The source code (filter_pushdown.cpp) explicitly avoids pushing filters into materialized CTEs, even when predicates could reduce CTE computation.

Query Torque opportunity: Detect filter/blocking-operator patterns and restructure queries to apply filters before windows, unnest operations, and CTE materialization.

Missing algebraic rewrites represent easy wins

DuckDB's 26 optimizer passes alibabacloud skip several transformations that external tools can apply:

Aggregation pushdown through joins is not implemented (Issue #17076). DuckDB won't transform:

```sql
-- DuckDB computes full join, then aggregates
SELECT SUM(sales) FROM orders JOIN customers ON ... GROUP BY customer_id
```

Into the equivalent eager aggregation:

```sql
-- Partial aggregation before join reduces intermediate size
SELECT SUM(partial_sum) FROM (
  SELECT customer_id, SUM(sales) as partial_sum
  FROM orders GROUP BY customer_id
) o JOIN customers c ON ...
```

Filter reordering by selectivity is limited. The REORDER_FILTER pass orders by computational cost, not by how many rows each filter eliminates. It doesn't use statistics for short-circuit evaluation.

DISTINCT elimination based on constraints is missing. DuckDB won't remove redundant DISTINCT when uniqueness is guaranteed by PRIMARY KEY or UNIQUE constraints.

Redundant join removal is limited. Joins that are provably unnecessary (e.g., joining on a foreign key where no columns from the referenced table are used) aren't automatically eliminated.

Query Torque opportunity: Implement eager aggregation, selectivity-based filter reordering, constraint-aware DISTINCT elimination, and dead join detection.

Runtime execution has predictable failure modes

Memory pressure cliffs: DuckDB spills to disk for hash aggregation, joins, and sorts, but several operations cannot spill and risk OOM:

- `PIVOT` operations (uses list() internally)
- Holistic aggregates: `median()`, `percentile_cont()`, `mode()`
- ART index creation (must fit entirely in RAM)
- Some window function auxiliary structures

Parallelism thresholds: DuckDB requires at least `threads × 122,880 rows` for parallelization. Small tables and point queries run single-threaded. For network-bound queries (S3/HTTP), the synchronous I/O model means over-provisioning threads (2-5x CPU cores) improves throughput.

Index limitations: DuckDB lacks traditional B-tree indexes. ART indexes only trigger at **<0.1% estimated selectivity** and support only equality/IN predicates on single columns. Zonemaps fail when data is randomly distributed within row groups.

Query Torque opportunity: Estimate memory requirements and suggest breaking queries into subqueries. Detect parallelism-limiting patterns. Recommend explicit `SET threads` for I/O workloads.

Community-documented anti-patterns the planner misses

Real-world cases where manual rewrites dramatically outperformed DuckDB's planner:

Anti-Pattern	Original Performance	After Rewrite	Issue
Loop of point lookups	3 iter/sec	300 iter/sec	Discussion #2368
TPC-DS Q37 join ordering	1.5 hours	5 minutes	Issue #3525
ENUM in JOIN condition	BLOCKWISE_NL_JOIN	HASH_JOIN	Issue #10387
IS NULL in WHERE (v1.3+)	4x slower	Baseline	Issue #18558
Window + external filter	Full table scan	Partition pruned	Issue #10352

ENUM type constraints in JOIN conditions trigger nested loop joins instead of hash joins. Moving the ENUM comparison from ON to WHERE restores optimal join selection.

IS NULL performance regressed in v1.3+. Moving IS NULL checks to FILTER clauses restores performance:

sql

```
-- Slow in v1.3+
SELECT ... WHERE t2.error IS NULL GROUP BY ...
-- Fast
SELECT MAX(data) FILTER (WHERE t2.error IS NULL) ... GROUP BY ...
```

GitHub

String vs. integer grouping: Aggregating on string columns is significantly slower than aggregating on integer keys and joining for labels afterward.

CTE and window function handling has exploitable gaps

CTE materialization changed in v1.3+ (PR #17459), defaulting to materialization instead of inlining. GitHub This yielded **1.2x to 2000x speedups** on CTE-heavy queries. github However:

- Predicates still cannot push into materialized CTEs
- Volatile functions (`random()`) may still inline incorrectly GitHub
- Materialized CTEs can be slower than equivalent temp tables (Issue #10451)

Window function combination is partial. DuckDB groups windows by common PARTITION BY/ORDER BY, but:

- Separate segment trees are built even for identical underlying data
- Frame-ordered non-aggregates require duplicate ORDER BY specification
- Streaming window evaluation is limited to simple patterns

Correlated subquery decorrelation is comprehensive but not complete. DuckDB uses the "Unnesting Arbitrary Queries" algorithm, but the ANY operator with correlated GROUP BY fails decorrelation (Issue #2999), and scalar subqueries returning multiple rows don't error (differs from Postgres/MySQL).

The attack surface summary for Query Torque

Category	Severity	Detection Signal	Rewrite Strategy
Join ordering failures	HIGH	>3 joins, large cardinality differences	Reorder by ascending cardinality
Missing aggregation pushdown	HIGH	GROUP BY after JOIN	Eager aggregation before join
Window blocking filter pushdown	MEDIUM	WHERE after window on PARTITION BY column	Pre-filter in CTE before window
UNNEST/LATERAL filter blocking	MEDIUM	WHERE after CROSS JOIN UNNEST	Subquery with filter first

Category	Severity	Detection Signal	Rewrite Strategy
CTE predicate pushdown	MEDIUM	Filter on materialized CTE columns	Inject filter into CTE definition
Cardinality estimation failures	MEDIUM	Joins on partitioned data, skewed columns	Restructure to reduce join size
ENUM in JOIN conditions	LOW	ENUM type comparison in ON clause	Move to WHERE clause
String GROUP BY inefficiency	LOW	GROUP BY on varchar columns	Aggregate on integer keys, join for labels

The highest-impact opportunities cluster around **join ordering**, **aggregation pushdown**, and **predicate pushdown through blocking operators**. These patterns are detectable from query structure alone without runtime statistics, making them ideal targets for LLM-based rewriting.

Conclusion

DuckDB's optimizer is sophisticated for an embedded OLAP database but exhibits systematic gaps that create a clear optimization surface for Query Torque. The most exploitable weaknesses are the **cost model's failure to accumulate intermediate join costs**, the **absence of histograms and correlation statistics**, and the **hard stops on predicate pushdown at window/UNNEST/CTE boundaries**. An external optimizer with access to query structure and optional runtime statistics can reliably detect and patch these gaps, with documented performance improvements ranging from 2x to 1000x on affected query patterns.