QueryTorque: VLDB Metrics Target Sheet

Exact numbers to hit, how they're calculated, and why each one matters

1. Primary Performance Metrics

These are the headline numbers for the paper. Every metric has an exact definition so there's no ambiguity about what's being measured.

Metric	Exact Definition	Competitor Baseline	Your Target	Why This Number
Geometric Mean Speedup (Postgres)	GM = (∏ speedup_i)^(1/n) where speedup_i = original_time_i / rewritten_time_i for each query i, across all n queries in the benchmark. Median of 3 runs per query.	**LITHE: 13.2x** *(on Hefty subset only, ~20-30 of 99 TPC-DS queries)* **R-Bot: ~5x** *(full TPC-DS suite)*	**≥15x** on full suite	Beats LITHE's 13.2x (which was cherry-picked). Running on the full suite makes the comparison honest and harder to dismiss.
Geometric Mean Speedup (DuckDB)	Same formula as above, run on DuckDB engine. TPC-DS 1TB. Same query set.	**R-Bot: ~1x expected** *(Postgres corpus doesn't transfer)* **LLM-QO: N/A** *(no hint injection in DuckDB)*	**≥10x**	Even a modest number here is a win because competitors score ~1x or N/A. This is the generalization proof.
Median Latency Reduction (DSB)	Median of per-query latency reduction across DSB queries. Reduction = 1 − (rewritten_time / original_time). Reported as percentage.	**LLM-QO: 68%** *(DSB on Postgres, physical plan steering)*	**≥75%**	DSB's skew makes cost estimators unreliable, which is where reasoning shines. Even matching 68% is acceptable if DuckDB numbers are strong.
Total Benchmark Runtime Reduction	1 − (sum of all rewritten query times / sum of all original query times). Single number across entire benchmark. Reported as percentage.	*Often not reported* *(competitors prefer GM because a few big wins inflate it less)*	**≥80%**	Report this alongside GM. Total runtime is what a DBA actually cares about. Shows practical impact, not just per-query statistics.

2. Query Coverage & Regression Targets

Reviewers look at regressions first. A system with 50x speedup on 10 queries but regressions on 5 will score lower than a system with 8x speedup on 70 queries and zero regressions. Breadth and safety matter more than peak performance.

Metric	Definition	Target	Reviewer Implication
Queries Improved	Count of queries where rewritten_time < 0.95 × original_time (i.e., at least 5% faster, outside noise margin).	**≥70 / 99** TPC-DS queries	Shows breadth. If you only speed up 10 queries by 100x, GM looks great but reviewers will see through it. 70+ means the architecture is general, not query-specific.
Queries Unchanged	Count of queries where rewritten_time is within ±5% of original_time. QueryTorque chose not to rewrite (no opportunity) or the rewrite was equivalent.	**≤20**	Acceptable. Some queries are already well-optimized. Reporting these honestly shows the system knows when not to intervene.
Regressions	Count of queries where rewritten_time > 1.05 × original_time (i.e., more than 5% slower). This is the number reviewers will look at first.	**0** (hard target)	Zero regressions is the gold standard. Even 1-2 regressions is defensible if explained, but >5% regression rate will draw a reject. If a rewrite is slower, fall back to the original — build this into the system.
Max Regression Severity	Worst-case slowdown on any single query: max(rewritten_time_i / original_time_i) across all queries. Only relevant if regressions > 0.	**<1.05x** (5% max)	A single query that's 2x slower will dominate the review discussion regardless of everything else. Implement a fallback: if the rewrite is slower on validation, use the original.
Semantic Correctness	Percentage of rewrites that produce identical result sets to the original query. Verified by comparing sorted output rows.	**100%**	Non-negotiable. A single incorrect result set means the system is unusable in production. This is pass/fail — anything less than 100% is a reject.

3. Overhead & Efficiency Metrics

This is where you make the "reasoning-first" claim concrete. If you say LITHE wastes tokens, show the numbers. If you say zero DB round-trips, measure it.

Metric	Definition	Target	Why It Matters
Optimization Wall-Clock Time	Median time from receiving the original SQL to producing the rewritten SQL. Excludes query execution. Measured end-to-end including all LLM inference.	**<30s median**	If the original query runs in 5s and you spend 60s optimizing it, the practical value disappears. Target: optimization time << original query runtime for the queries you improve.
Amortization Ratio	optimization_time / time_saved_per_execution. Number of executions needed before the optimization pays for itself. E.g., if optimization takes 30s and saves 60s per run, ratio = 0.5 (pays off on first run).	**<1.0** (pays off in 1 run)	Analytical queries typically run repeatedly (dashboards, reports). If amortization ratio <1, the optimization is free after the first execution. Report this per query.
Tokens per Query	Total input + output tokens consumed by the LLM to produce the rewrite. Sum across all	**Report actual** (compare to LITHE)	You claim LITHE wastes tokens in its loop. Prove it. If LITHE uses 50k tokens over 15 iterations and you

	reasoning steps/agents in the pipeline.		use 8k in 1 pass, that's a 6x efficiency gain. This is a concrete version of your "reasoning-first" claim.
DB Round-Trips	Number of times the system queries the database during optimization (e.g., EXPLAIN calls, statistics lookups, validation runs). Zero means fully offline optimization.	**0–1** (ideally 0)	LITHE needs multiple round-trips per iteration (syntax check + cost estimate). LLM-QO needs the planner. If you need 0 round-trips to produce the rewrite, that's a fundamental architectural advantage and directly enables cross-engine portability.
Dollar Cost per Query	API cost to optimize one query at current model pricing. tokens_used × price_per_token. Report for your chosen model.	**Report actual** (context for practitioners)	Not strictly required for VLDB, but practitioners reading the paper will want this. If optimization costs $0.03 and saves 10 minutes of compute, the ROI is obvious.

4. DSB Spotlight Queries (Skew Cases)

These three queries are where you demonstrate that reasoning about data characteristics beats statistical estimation. For each query, briefly explain the skew pattern in the paper so reviewers understand it's principled analysis, not cherry-picking.

DSB Query	Why It's Hard (Skew Pattern)	What Competitors Do	Your Target & Narrative
Q21	Correlated predicates across dimension tables. The optimizer assumes independence between filters, producing cardinality estimates that are orders of magnitude off. Joins explode because the planner picks nested loops expecting few rows.	Cost estimator misleads LITHE's loop. R-Bot has no rule for correlated predicates (not in manuals). LLM-QO's plan steering can't fix the cardinality error, only the join order.	**≥20x per-query speedup.** Show that reasoning identifies the correlation and rewrites to avoid the bad join. Explain the skew pattern in the paper.
Q36	Heavy aggregation with skewed group-by keys. A small number of groups contain most of the data. The optimizer allocates memory and chooses algorithms assuming uniform distribution, causing spills and hash table resizing.	Standard optimizers choose hash aggregation sized for the average case. Spills to disk on skewed groups. No competitor addresses this at the logical level.	**≥10x per-query speedup.** Show the rewrite restructures the aggregation to handle the skew pattern. This is a reasoning win, not a plan-steering win.
Q78	Multi-way join with selective predicates on skewed columns. Filter selectivity varies dramatically by value. The optimizer uses average selectivity, leading to wrong join ordering and materialization strategy.	LITHE may eventually find a better plan by luck but wastes iterations. R-Bot has no data-aware rules. LLM-QO can adjust join order but can't push filters or restructure the SQL.	**≥15x per-query speedup.** Show reasoning identifies which predicates are selective on the skewed distribution and restructures accordingly.

5. Cross-Engine Scorecard

This is the table that goes in the paper. Side-by-side, same benchmark, two engines. The generalization gap should be visually obvious.

Metric	QT (Postgres)	QT (DuckDB)	R-Bot (Postgres)	R-Bot (DuckDB)
GM Speedup	**≥15x**	**≥10x**	~5x	~1x (expected)
Queries Improved	≥70/99	≥60/99	~40-50/99	~0-5/99
Regressions	**0**	**0**	Report actual	Possible regressions
Correctness	**100%**	**100%**	Report actual	Report actual
Porting Effort	N/A (native)	**~2 hrs rule writing**	N/A (native)	New corpus needed

Key insight: The DuckDB column doesn't need to be as strong as Postgres. Even 10x on DuckDB vs ~1x for R-Bot is a decisive generalization win. The porting effort row (2 hours of rule writing) is the practical proof that cross-engine portability is real, not theoretical.

6. How to Read These Targets

Minimum viable paper vs strong paper:

- **Minimum viable:** ≥8x GM on Postgres full suite (beats R-Bot clearly), ≥5x on DuckDB (proves generalization), 0 regressions, 100% correctness. This gets you to revision.
- **Strong paper:** ≥15x GM on Postgres (beats LITHE), ≥10x on DuckDB, ≥70 queries improved, overhead <30s, token efficiency 5x+ better than LITHE, DSB spotlight queries showing 15-20x. This gets you accepted.
- **Standout paper:** All of the above, plus a third engine (e.g., Snowflake or SQLite), plus a failure case analysis where you honestly show where QueryTorque doesn't help and explain why. This gets you a best paper nomination.

What reviewers check in this order:

- 1. Correctness (100% or reject). 2. Regressions (any unexplained = major revision). 3. Breadth of improvement (queries improved count). 4. Headline speedup (GM). 5. Generalization evidence. 6. Overhead analysis. **Design your experiments in this priority order.**