Verified LLM SQL Query Rewrite

Final Presentation - Prashanth, Ruiqi, Melody

Main Goals

1. Prompt Engineering: give LLM background about the table before query
   - E.g. schema, statistics
   - What table information do we need to give the LLM for it to best optimize the query?
2. Generate data for various tables to test for equivalence
   - Goal is to be as exhaustive as possible, maybe start with small → bigger / more complex
   - Sorted values, NULL values, different scenarios
   - If not the same output, give counterexamples
3. Test out different LLMs to compare optimization performance
   - How do performances vary between ChatGPT, Gemini, Claude, etc?

Our Progress

Building the verifier pipeline

- Integration with QED => bad performance
- Benchmarking verifier correctness
- Robust data generation mechanism on different data distribution

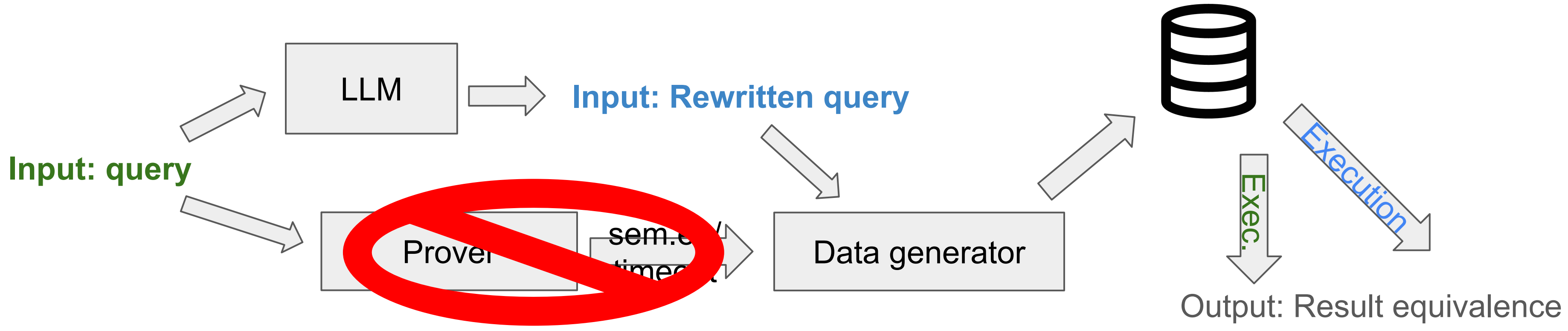
Testing how good is the LLM at optimizing queries

- Randomizing table + column names to avoid LLM prior knowledge
- Benchmarking LLM optimization performance

Verifier Pipeline

New Architecture (standalone prototype)

1. Give LLM background information about the table (schema and statistics)
2. Feed the original query to the LLM.
3. Ask LLM to generate new query (optimized, shortening the SQL etc.)~~ and output query from LLM into semantic prover (QED)~~
4. Generate datasets to run the queries on & check for identical outputs



No QED

- Does not support a lot of keywords & short timeout period
  - ex. only supports “left outer join” instead of “left join”
- In testing with DSB & SlabCity, mostly gets “cannot be determined”
  - more so for DSB, likely because of more complex
- QED says the following 2 queries are not equivalent:

```
SELECT e.ename
FROM employees e
INNER JOIN departments d
ON e.deptno = d.deptno
WHERE d.deptname = 'Sales';
```

```
SELECT e.ename
FROM employees e
WHERE e.deptno IN (
  SELECT d.deptno
  FROM departments d
  WHERE d.deptname = 'Sales'
);
```

Verifier Pipeline

- **prompt_llm.py** prompts the LLM to give optimized queries by giving it the original queries, table schemas, and statistics on the table
    - statistics: table cardinalities, generate on the fly
- **generate_data.py** generates random tables based on the scheme given for testing query equivalence
    - Using selectivity to generate small synthetic data on the original schema
    - Higher probability density on boundary values
    - Pattern match “column = xxx” to make sure xxx appears in synthetic data
- **check_equivalence.py** combines the results from above and outputs whether the original query and the LLM-optimized query are equal based on QED and the random tables generated

- Testing correctness: unit tests, DSB, SlabCity

Strength of our implementation

- Minimal overhead, can exploit DuckDB optimizer with fuzzing
  - Significantly faster than QED baseline
- Try out different data distribution to cover skewed and corner cases
- LLM produced different queries when schema names are replaced with random adjectives, separating LLM prior context knowledge with prompted knowledge
- Only feed relevant stats for current query to reduce input tokens and API cost
- Only need to pre-process the queries & generate synthetic data once for the same workload

Area of Improvements

- Following the data distribution isn't enough
  - Some queries will have specific predicates (e.g. LIKE %%, DATE=xxx)
  - Cannot be easily pattern matched
  - If we are unable to generate data satisfying the predicates, all queries will return no data, creating false positives
  - Correlations between different columns
- Generating all synthetic data for a workload upfront
  - Doesn't account for changing data distribution
- Cannot quantify performance improvement efficiently because tail latency for LLM optimized query very high

Assessment on quality and correctness

- Verifier metric: False Positives + False Negatives
- LLM rewrite metrics: performance improvement/degradation

Workload:

- Plan level optimization: TPC-DS, DSB
- Query level optimization: SlabCity (LeetCode)

Benchmarking

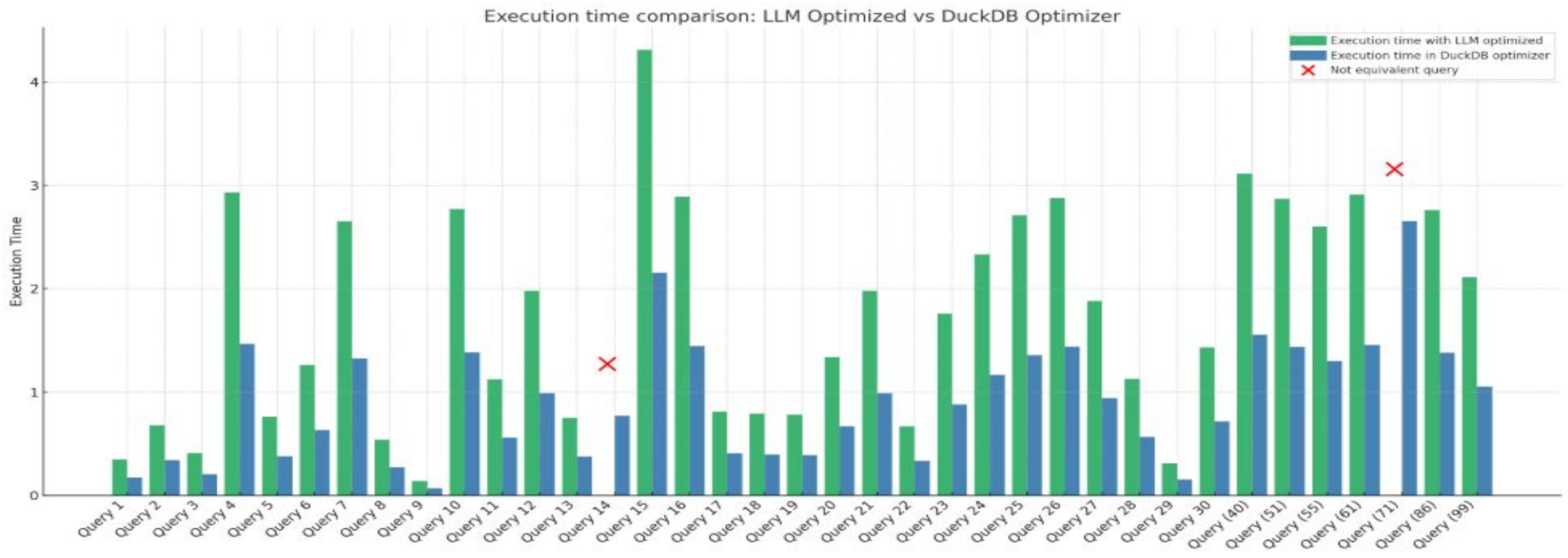
Prompt Engineering

“Your job is to rewrite SQL queries to optimize performance in DuckDB. Make sure it has the same output and doesn't modify the predicates even if it seems wrong. Only output the optimized query in one line, don't include any other additional words and newline characters. You are given the following workload stats (cardinalities + selectivity) and schema to help you with rewriting the queries: [original query]”

- Need to specifically mention “same output” and “not modify predicates”
- Need to specifically mention “DuckDB” for query compatibility
- Give LLM cardinalities + selectivity

TPC-DS (when LLM knows it's TPC-DS queries)

Execution Time Comparison: LLM Optimized Vs DuckDB Optimizer



TPC-DS (Environments) and features

Ran on the following configuration

1) Ubuntu on AWS
   - 64 Gb Disk Size
   - C5a.8x Large processor
   - 64GB Memory
2) DuckDB
3) Ran with following settings
   - Non Optimized Query with DuckDB Optimizer Off
   - LLM Optimized Query with DuckDB Optmizer Switched Off
   - DuckDB Optimized Query
4) TPCDS Queries
   - Scale Factor 1
   - 1 to 30 queries
   - Query 40, 51, 55, 61, 71, 86, 99 (categorized as complex queries)
5) LLM - ChatGPT 4o

TPC-DS

- Between Queries 1-30
  - 11 queries gave false equivalent output in the first attempt and had to re prompt the LLM to produce a new query.
  - 5 queries never gave right output despite prompting 3-4 times (query 14 and query 71)
- Underperformed than estimated
- Times vary between 0.2 seconds to 5 seconds overall
- For few queries, query equivalence API was unable to detect if the queries are equivalent - used our logic to check equivalence.

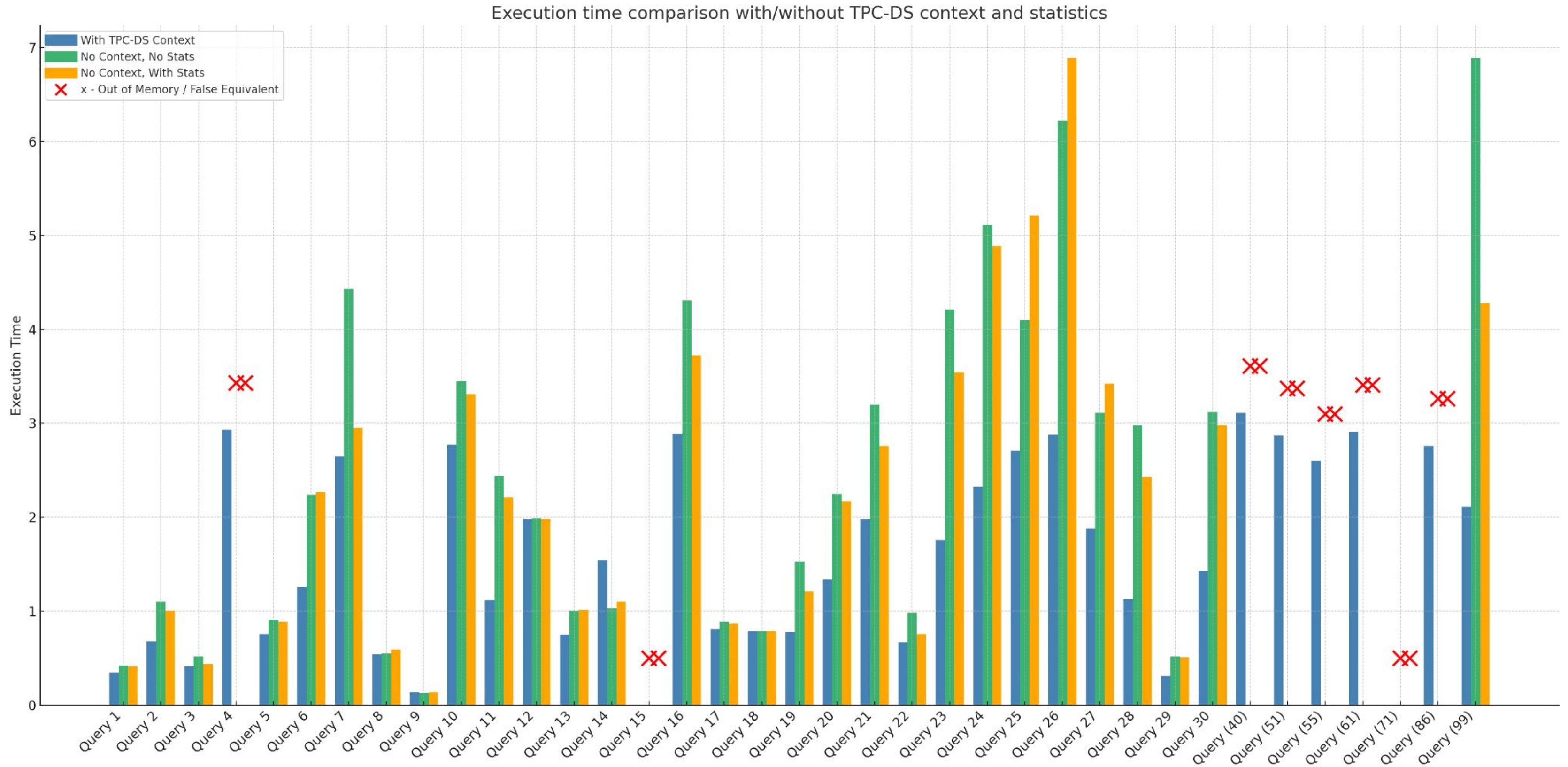
Anonymized TPC-DS (LLM doesn’t know it’s TPC-DS queries and data)

- Names of Tables: Random three letter word (for eg: abc)
- Column names: Table Name + Index number (for eg: abc1)
- SQL query parsed using library (sqlglot), replaced all names of tables, column and aliases.
- Write small grammar to save Postgres specific dialects.
- Using the queries to generate LLM optimized queries via API tokens for ChatGPT.
- Performed 2 types - one with context and one without context

Old Results vs New Results - Performance

When it has the context	When it doesn’t have the context
5 False equivalent queries (in first attempt)	11 False Equivalent queries (in first attempt)
2 queries never gave right output (3-4 prompt)	5 queries never gave right output (3-4 prompt attempts)
2 queries out of memory	3 queries out of memory

Overall benchmarks



Anonymized DSB

Before changing up the column and table names

- In ~85% of the queries sampled, the original version times out (>30s) without the DuckDB optimizer, whereas the LLM-optimized query executes successfully in comparable time to DuckDB-optimized version

After changing up the column and table names

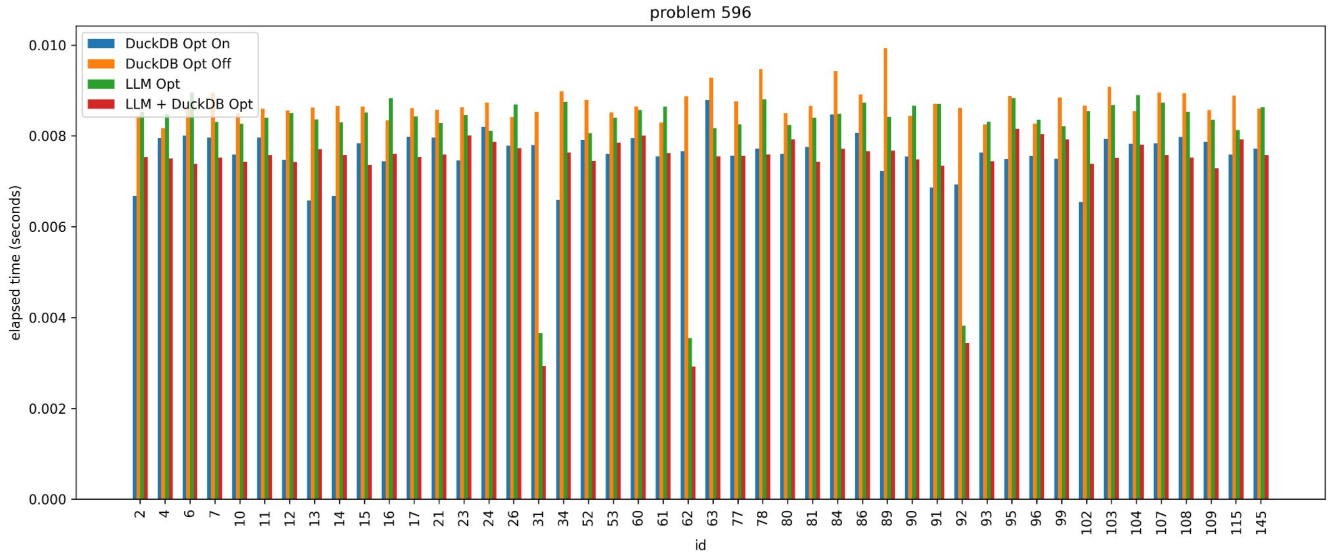
- Scale factor = 1, LLM = GPT 4o
- QED accuracy: cannot determined any; Synthetic data accuracy: 92%
- LLM accuracy: 83% of the generated queries are equivalent to original
- 62.5% of the LLM-optimized queries timed out (over 3 minutes)
- Does not use CTEs at all

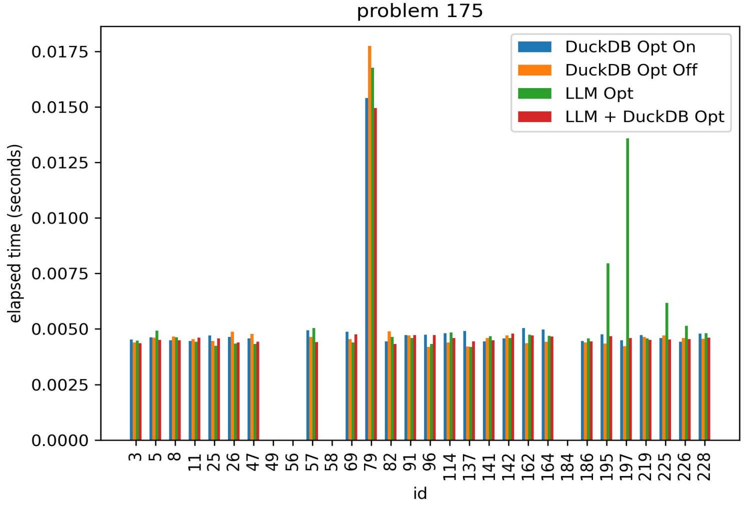
Summary: LLM does rely on having seen the DSB queries before to optimize

SlabCity benchmark

- Leetcode dataset, tested 1111 queries, LLM = GPT 4o
  - Skipped some due to our data generation script not supporting ENUM & other errors
- Average time measured by running each query 3 times (resetting in between)
- 4 settings: original query with DuckDB optimizer, original query without DuckDB optimizer, LLM optimized query with DuckDB optimizer, LLM optimized query without DuckDB optimizer

- QED accuracy: 56.0%
  - ~60% cannot be determined, categorized as not equivalent
- Synthetic data accuracy: 95.7%
  - Average F1 score: 97.2%
- LLM accuracy: 76.7% of the generated queries are equivalent to original





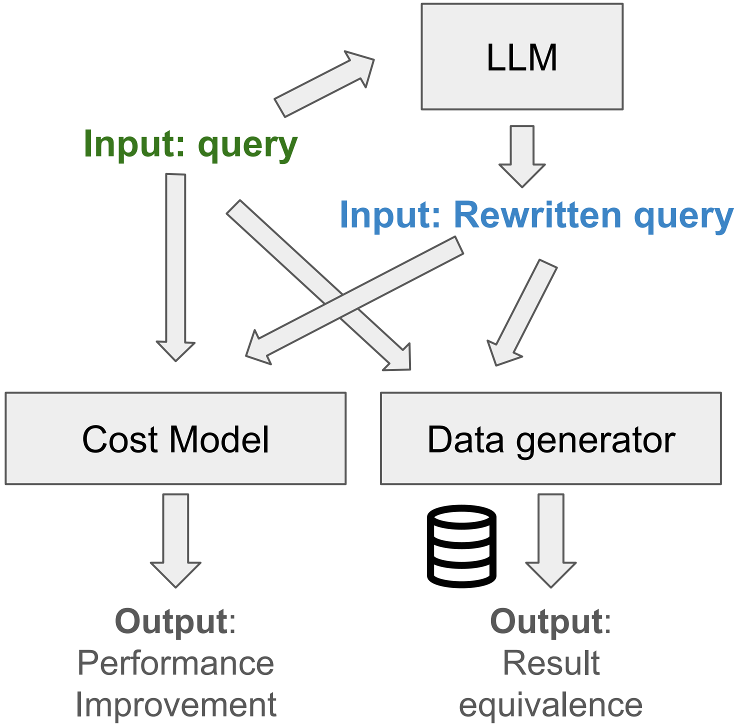
- Most queries have equivalent or better performance compared to having no DuckDB optimization
- More plots can be found here:
  - https://github.com/s-wangru/Verified-SQL-query-rewrite/tree/main/slabcity/results/plots

Concluding thoughts & Future Work

- LLM requires context / prior knowledge to generate good optimized queries
- When using a well-known benchmark, need to randomize the names

Future Work

- QED: play with timeout period & SQLGlot
- Data generation: account for better variety of data distribution
- Cost model: predict the execution time instead of actually running the queries
- LLM models: test out performances of different models & use failed test to prompt LLM for improvement


LLM
Input: query
Input: Rewritten query
Cost Model
Data generator
Output: Performance Improvement
Output: Result equivalence