Leveraging Query Optimizers to Verify Soundness of LLM-Based Query Rewrites for Real-World Workloads, and More!

Vivek Narasayya, Surajit Chaudhuri

Microsoft Corporation

Acknowledgments:

Kukjin Lee, Christian König, Nico Bruno, Vassilis Papadimos, Kaushik Rajan, Xiaoying Wang, Wentao Wu

Query Optimizer

- Query Optimizer (QO) is responsible to produce an efficient plan
  - for an arbitrary SQL query
  - using limited resources and time
  - with limited statistical information of the data

- QO produces good plans for most queries…
  - …but sometimes produces poor plans

- Examples of techniques used by DBAs and application developers to tune poorly performing queries
  - Query hinting, query rewriting

Query Hinting

- Query optimizers expose the mechanism of query hinting
  - Hints influence the plan chosen for the query
  - E.g., OPTION (FORCE ORDER) instructs optimizer to use the join order specified in the query

- Prior work that recommends which hint(s) to use for a particular query
  - Bao [SIGMOD 2021], AutoSteer [VLDB 2023], FASTGres [VLDB 2023], …
  - Query Hint Recommendation Tool for Microsoft SQL Server in Public Preview (Oct 2024)

Query Rewriting

- Real-world example of manual rewriting in one of Microsoft's services
  - Leads to faster execution since the execution plan for Q2 can use indexes more effectively
  - interaction_log is a VIEW which joins 45 tables

Q_1:

```
SELECT TOP (100)
   il.interaction_subject    AS subject,
   il.attempt_count          AS attempt_count,
   il.interaction_id         AS interaction_id

FROM interaction_log AS il

WHERE (
            il.person_uid = @person_uid
    OR
            il.interaction_id = @interaction_id
)
ORDER BY il.interaction_id ASC;
```

Q_2:

```
SELECT TOP (100) *
FROM (
   SELECT TOP (100)
      il.interaction_subject    AS subject,
      il.attempt_count          AS attempt_count,
      il.interaction_id         AS interaction_id
   FROM interaction_log AS il
   WHERE il.person_uid = @person_uid
   ORDER BY il.interaction_id ASC

   UNION

   SELECT TOP (100)
      il.interaction_subject    AS subject,
      il.attempt_count          AS attempt_count,
      il.interaction_id         AS interaction_id
   FROM interaction_log AS il
   WHERE il.interaction_id = @interaction_id
   ORDER BY il.interaction_id ASC
) AS subquery;
```

LLM-Generated SQL Query Rewriting

- Recent work on generating SQL query rewrites using LLMs to improve query performance

- GenRewrite [arXiv.org, 2025], LITHE [EDBT 2026]
  - Propose a set of carefully designed prompts:
    - “Replace the correlated subquery with a precomputed aggregate in a separate CTE”
  - Iteratively refine rewrite to reduce errors due to LLM hallucinations
  - Generate counter examples efficiently to identify when rewritten query is NOT equivalent
  - Largely focused on industry benchmarks

Key Questions

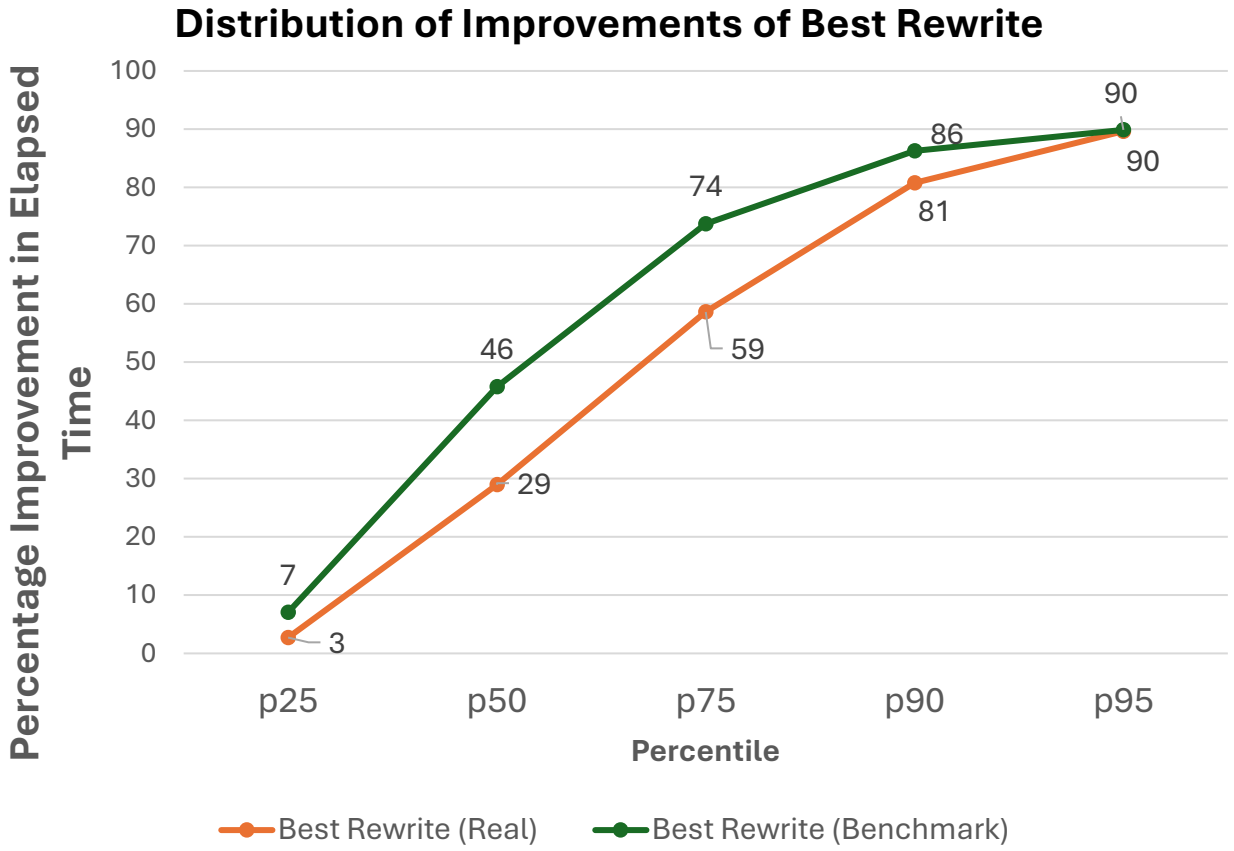
- Are LLM-generated rewrites effective on real-world queries?
  - On enterprise databases that LLMs have not been trained on
- How to verify if a rewritten query is semantically equivalent to the original query?
  - **Automation is crucial** to avoid “human-in-the-loop” tuning

Experimental Setup and Methodology

- Goal: compare quality of LLM-generated rewrites for real-world vs. benchmark queries
- Four Microsoft SQL Server customer workloads
  - Queries contain group-by, aggregation, nested sub-queries, CTEs, views, outer joins
- LLMs: GPT-4o-mini, GPT-5
  - Up to 3 invocations per model
- Prompts:
  - LLM is not constrained: **GENERAL**
  - **CTE**, **SUBQ**: constrained to remove redundant computation in query (LITHE)
  - **QHINT**: return query adorned with query hints

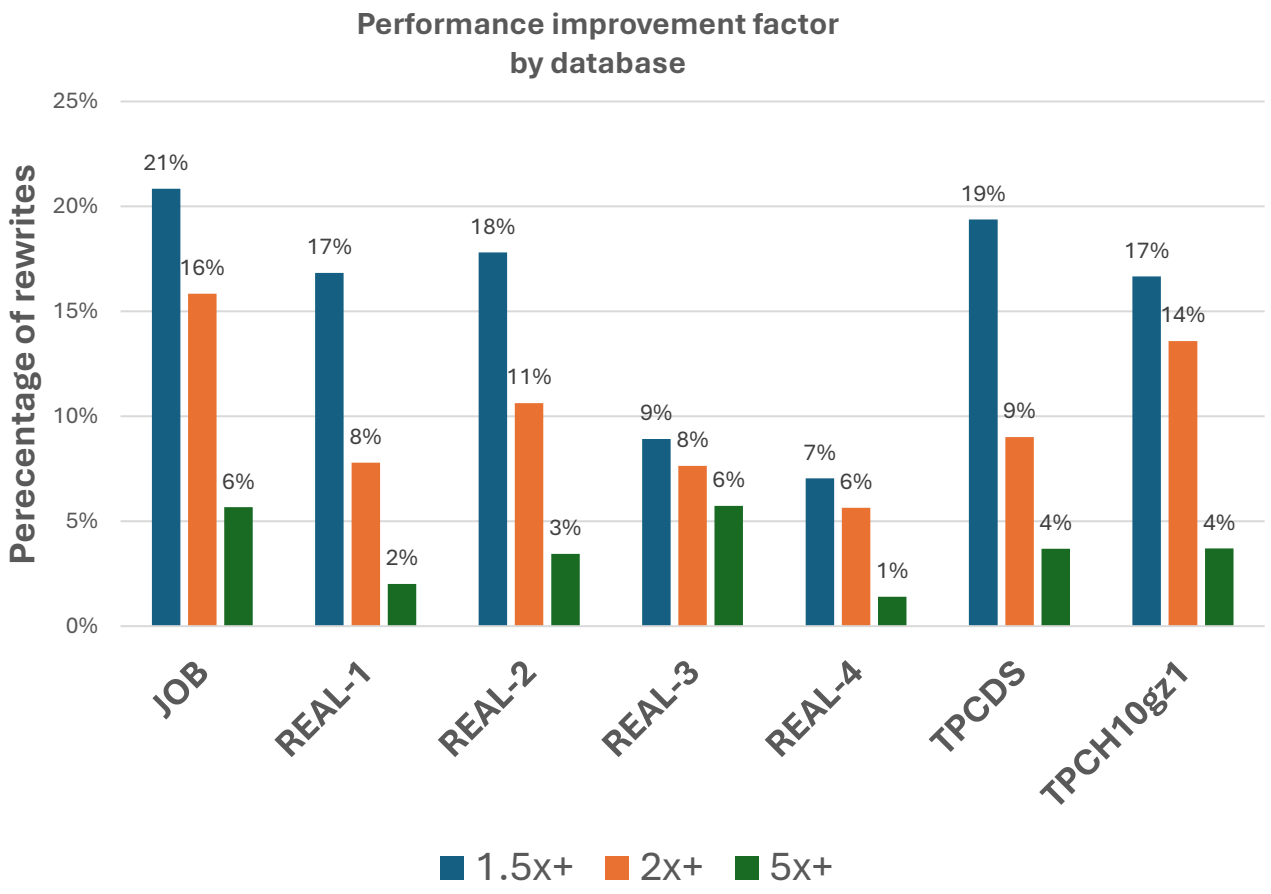
	Number of queries	Min, Max, Median number of tables in query
REAL-1	39	1, 8, 3
REAL-2	40	5, 28, 16
REAL-3	25	1, 8, 4
REAL-4	12	4, 14,10
TPC-DS	99	1, 13, 5
JOB	113	4, 15, 8
TPC-H	22	1, 7, 3

Performance Improvement of Best Rewrite for Query



- Restrict to “candidate” rewrites
  - Produce same results as original query on given database
- Median **real-world** query obtains 29% improvement

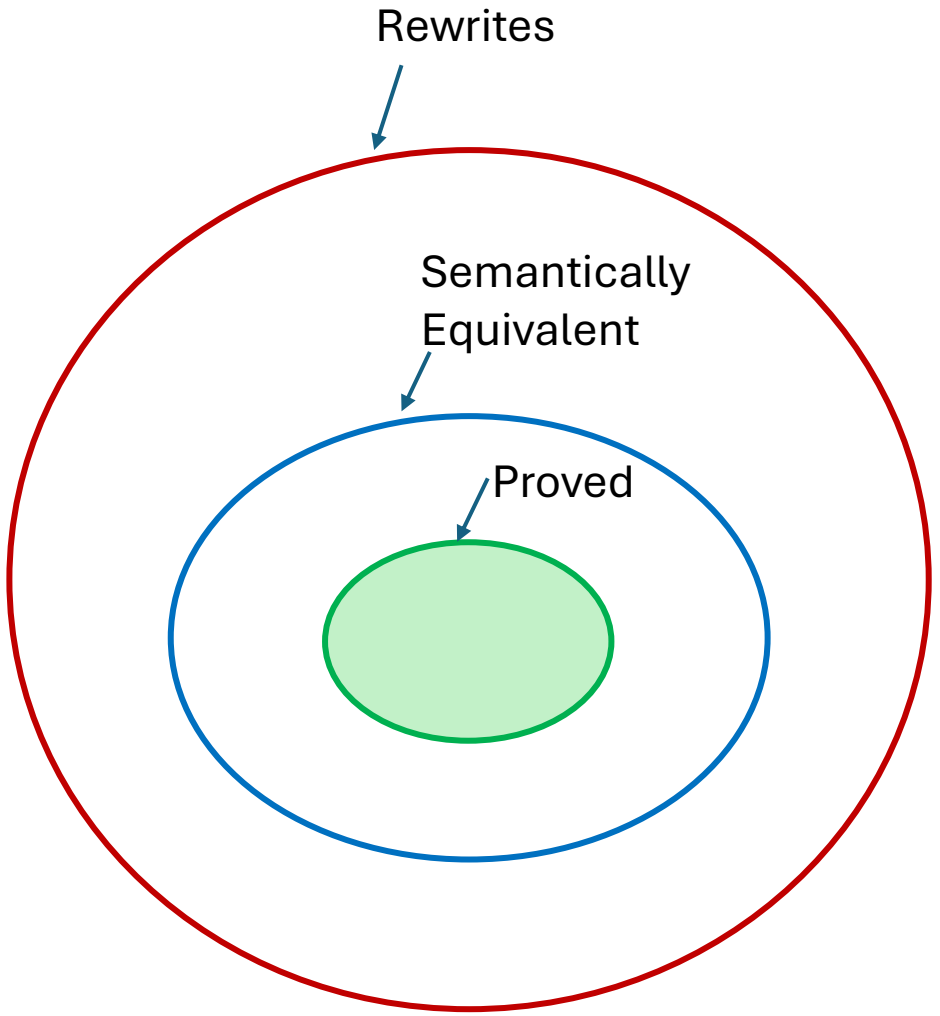
Distribution of Improvement across Databases



- Noticeable fraction of queries on each database see meaningful performance improvements
  - E.g., 17% of rewrites on REAL-1 improve by 1.5x or more

Automatically Verifying Semantic Equivalence of Two SQL Queries

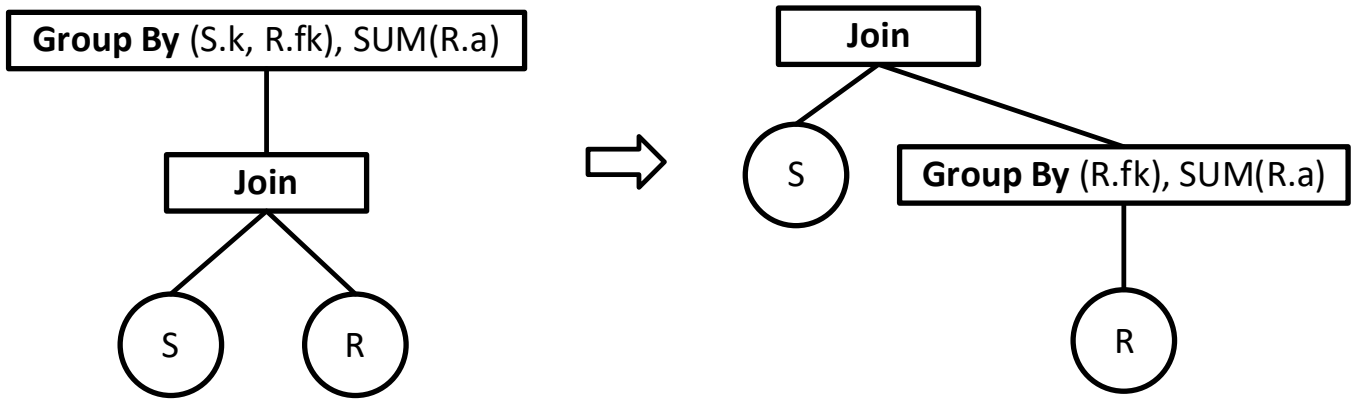
- Automation is **crucial:** do not want human "in-the-loop"
- Prior work: convert SQL queries to logic formulas, then use Satisfiability Modulo Theory (SMT) solvers to check if formulas are equivalent
- We find SQL equivalence checkers inadequate in practice
  - E.g., QED [VLDB 2024] is unable to verify any LLM-generated rewrites on real-world queries
  - Class of queries supported is limited, e.g., inability to handle aggregations



Idea: Re-purpose Query Optimizer for Verifying Semantic Equivalence

- Optimizers apply logical transformation rules to explore the search space
  - Developed over the decades to help optimize complex, real-world queries
  - Rules for queries with Group-By, Join, Nested sub-queries, CTEs, Views, …
  - Microsoft SQL Server optimizer has 325+ rules

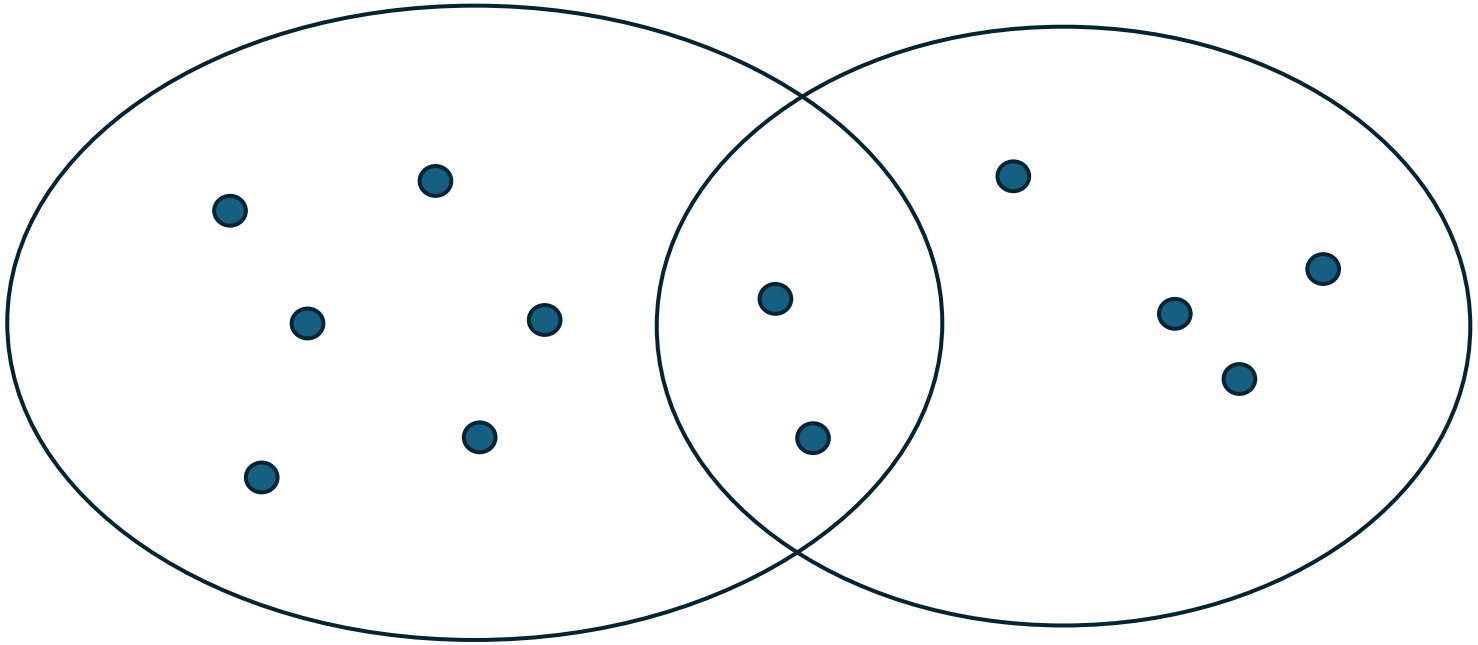
Group-By push down rule:



- Each rule application **preserves semantic equivalence**

Semantic equivalence checking using QO

Logical expressions enumerated for Q_1



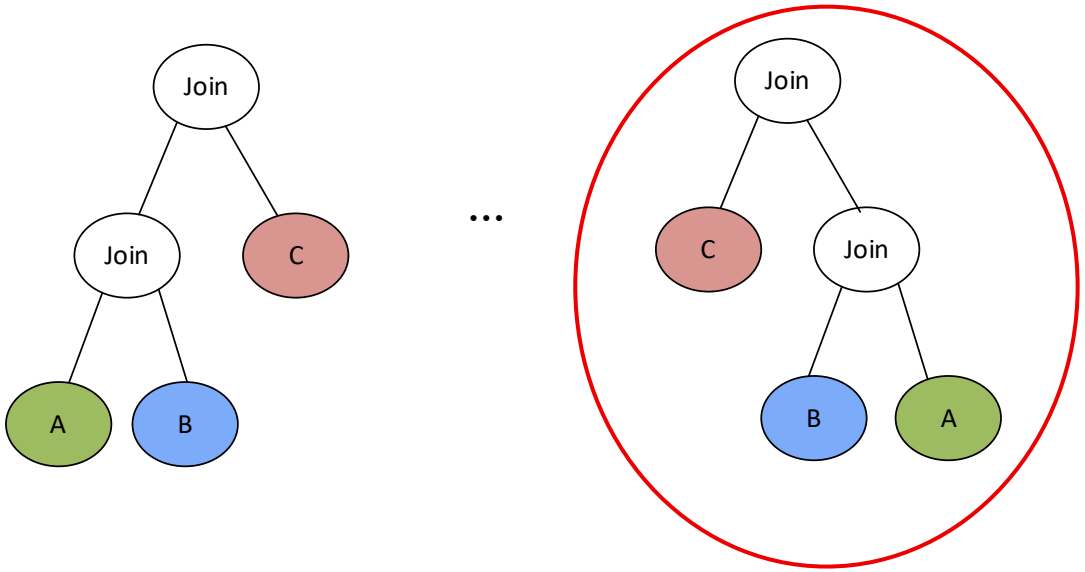
Logical expressions enumerated for Q_2

- If the logical expressions enumerated by the optimizer for Q_1 and Q_2 **share at least one logical query expression in common**, then Q_1 and Q_2 are equivalent

Memo data structure used by optimizers

Query:

```
SELECT  * FROM  A, B, C
WHERE   A.x = B.y AND B.p = C.q
```



Memo:

Groups	Expressions
Group 6: (root)	1: Join(4, 3) B.p=C.q; 2: Join(3, 4) B.p=C.q; 3: Join(5, 1) A.x = B.y; 4: Join(1, 5) A.x = B.y
Group 5:	1: Join(2, 3) B.p=C.q; 2: Join(3, 2) B.p=C.q
Group 4:	1: Join(1, 2) A.x = B.y; 2: Join(2, 1) A.x = B.y
Group 3:	1: Get (C)
Group 2:	1: Get (B)
Group 1:	1: Get (A)

- Compactly represents all logical expressions enumerated by QO

Example

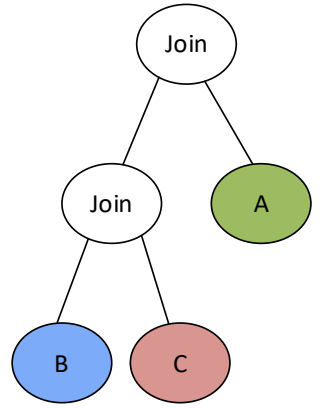
Q_1:

```
SELECT  *
FROM    A, B, C
WHERE   A.x = B.y AND B.p = C.q
```

Groups	Expressions
Group 6: (root)	1: Join(4, 3) B.p=C.q; 2: Join(3, 4) B.p=C.q; 3: Join(5, 1) A.x = B.y; 4: Join(1, 5) A.x = B.y
Group 5:	1: Join(2, 3) B.p=C.q; 2: Join(3, 2) B.p=C.q
Group 4:	1: Join(1, 2) A.x = B.y; 2: Join(2, 1) A.x = B.y
Group 3:	1: Get (C)
Group 2:	1: Get (B)
Group 1:	1: Get (A)

Q_2:

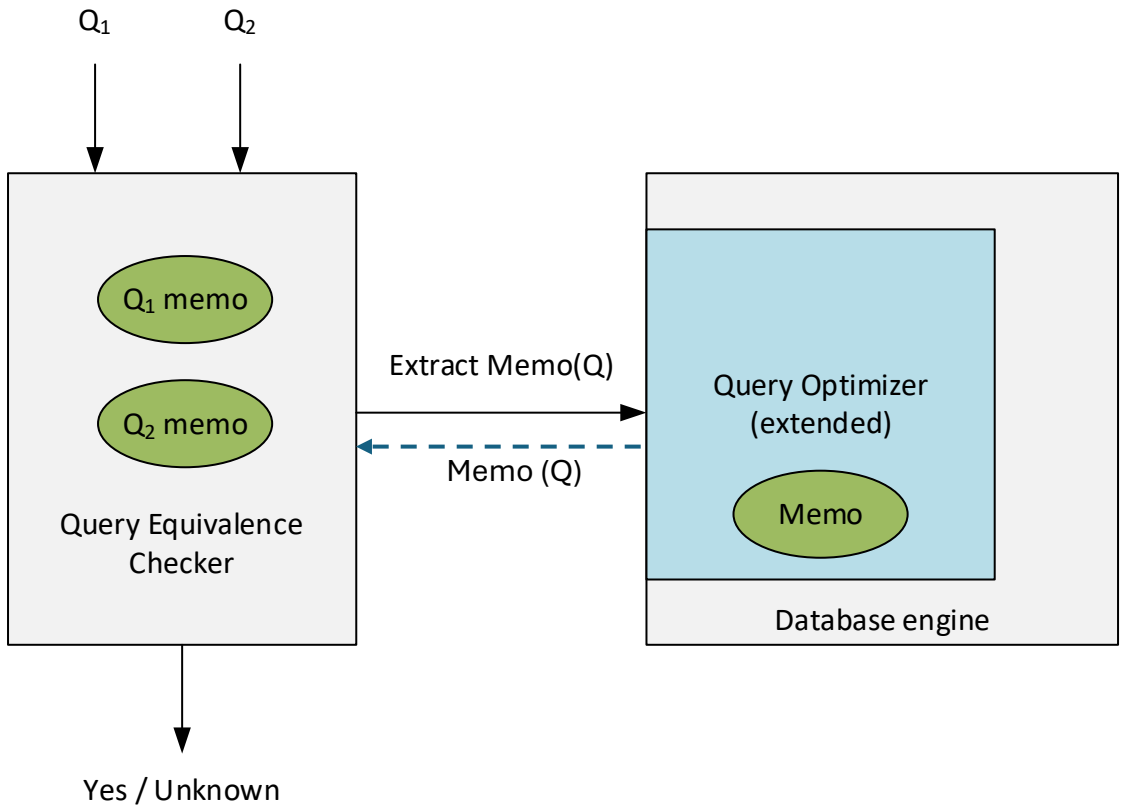
```
SELECT  *
FROM B INNER JOIN C on B.p = C.q
INNER JOIN A on B.y = A.x
```



Groups	Expressions
Group 5: (root)	1: Join(4, 3) B.y= A.x
Group 4:	1: Join(1, 2) B.P = C.q
Group 3:	1: Get (A)
Group 2:	1: Get (C)
Group 1:	1: Get (B)

QO-Verify: Mechanism for verifying equivalence using the Query Optimizer

- Input: SQL queries Q_1 and Q_2
- Output:
  - **Yes**, if queries are equivalent
  - **Unknown**, otherwise
- Disable cost-based pruning when invoking query optimizer
- Efficient, depth-first search algorithm starting at root group of each memo
  - See paper for additional details on efficiency and coverage

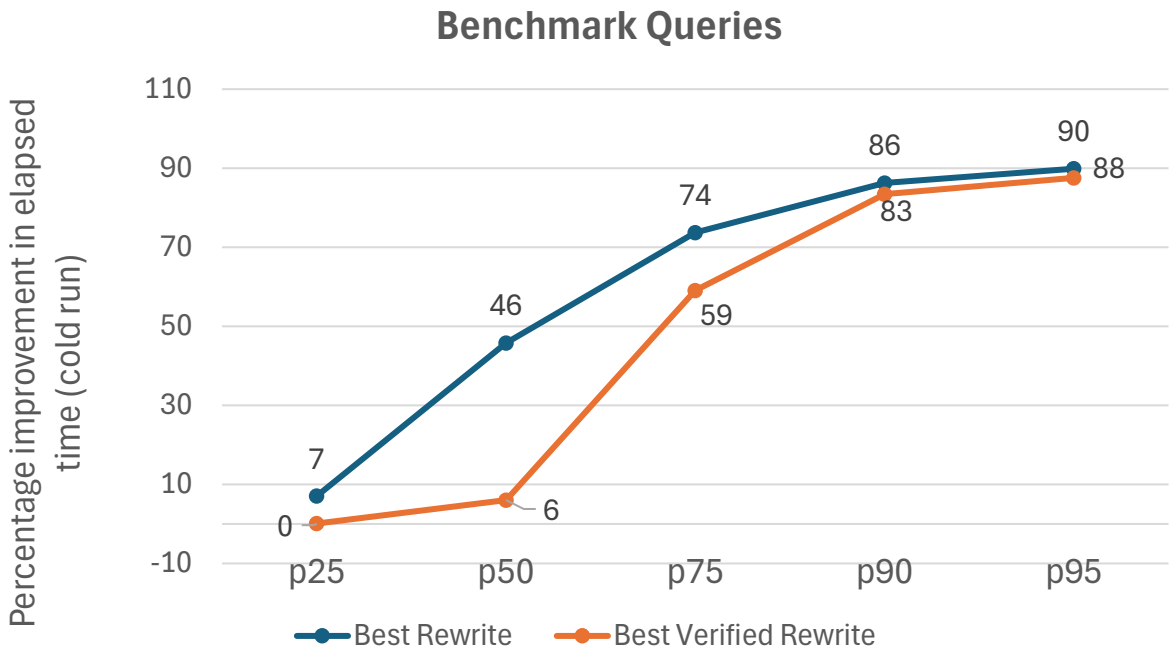


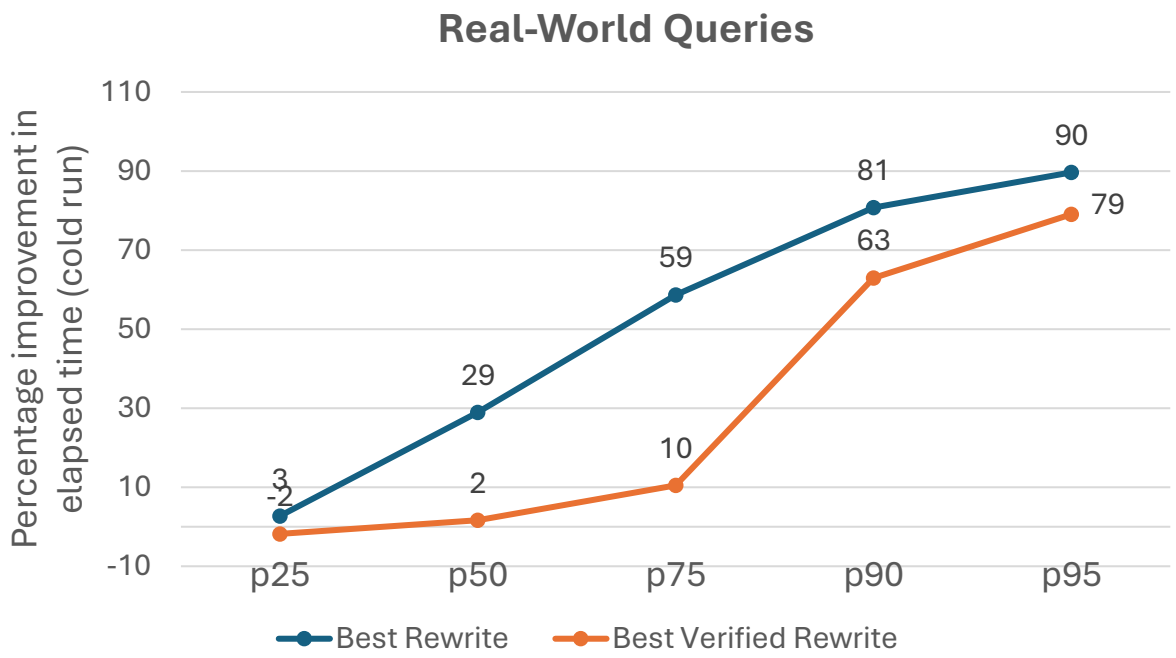
Coverage and Efficiency

- 3100+ **candidate rewrites**
  - Results of query and rewrite are identical on given database
- 37% of candidate rewrites were verified using QO-Verify
- A likely reason coverage is not higher is that one or more transformation rules are missing in the QO

Overheads	Median	P90
Time to obtain memo	0.57 sec	17.5 sec
Equivalence checking	0.03 sec	5.4 sec
Memory	0.33 MB	4.4 MB

Performance Improvement of Best Verified Rewrite for Query



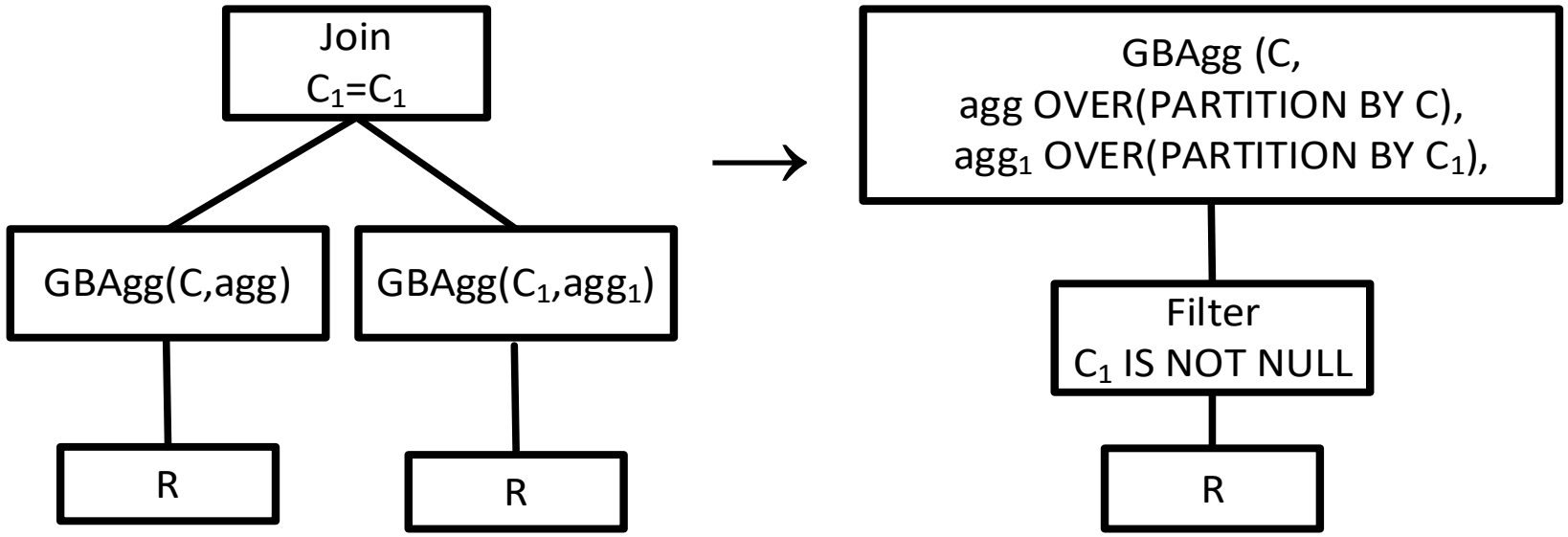


- Noticeable improvements even with verified rewrites only
- Step forward compared to alternative of no verified rewrites!

Can LLM-generated rewrites enrich the Query Optimizer?

- There are many candidate rewrites whose performance much better, but QO-Verify returns **Unknown**
- Some of these candidate rewrites are actually semantically equivalent, which a query optimizer developer could verify **manually**
- QO developer manually discovers a beneficial candidate transformation rule that is **missing in their query optimizer**
- If the rule can be proved to be correct, QO developers can potentially add the rule to the query optimizer
  - May benefit users of the database engine broadly

Candidate Transformation Rule



- Join of two subqueries that are identical but differ in their group-by columns
  - Group-by columns of one subquery is a subset of the other
- Rule broadens coverage of prior optimizations
  - Generalized Sub-Query Fusion for Eliminating Redundant I/O from Big-Data Queries. **OSDI 2020**
  - Computation reuse via fusion in Amazon Athena. **ICDE 2022**

Summary

- LLM-generated rewrites show promise in improving performance of **real-world queries**
- **Automated verification** of rewrites is crucial for adoption of LLM-generated rewriting in practice
  - Not practical to require “human-in-the-loop”
- QO-Verify: **Query Optimizer-based semantic equivalence checker** is the first technique that allows verifying rewrites of complex, real-world SQL queries
- Query optimizer developers may be able to **discover** valuable **transformation rules** to add to their query optimizer using LLM-generated rewrites

Open Problems

- Improving equivalence checking of SQL queries
- Developing efficient testing-based techniques, to prove that two queries are **not** equivalent
- Automatically identify candidate transformation rules from LLM-Based query rewrites

Questions? Comments?

Backup Slides

QO-Verify Algorithm

- Leverages memo for efficiently searching for a common logical expression between Q_1 and Q_2
- Top-down search algorithm starting at root group of each memo
- Efficiency optimizations
  - Early termination, Caching, signatures

Algorithm 1 QO-Verify: Optimizer-based Query Equivalence

Input: Two queries Q_1, Q_2.
Output: `true` if equivalent, else `unknown`.

```
1:  function EQUIVALENT(Q1, Q2)
2:      M1 ← GETMEMO(Q1)
3:      M2 ← GETMEMO(Q2)
4:      return MATCHGROUPS(M1.root, M2.root) ? true : unknown
5:  function MATCHGROUPS(g1, g2)      ▷ Equivalent if any pair of
    expressions, one from each group, are equal
6:      for all e1 ∈ g1.exprs do
7:          for all e2 ∈ g2.exprs do
8:              if MATCHEXPRS(e1, e2) then
9:                  return true
10:     return false
11: function MATCHEXPRS(e1, e2)
12:     if e1.root ≠ e2.root or |e1.children| ≠ |e2.children| then
13:         return false
14:     for i ← 1 to |e1.children| do
15:         if not MATCHGROUPS(e1.children[i], e2.children[i])
    then
16:             return false
17:     return true
```

Other Aspects of Empirical Study (see paper)

- Variation across:
  - Databases
  - Prompts
  - GPT-4o-mini vs. GPT-5
- How many calls to LLM are required to obtain a rewrite with significant (e.g., 2x) performance improvement?