SlabCity: Whole-Query Optimization using Program Synthesis

Rui Dong*
University of Michigan
ruidong@umich.edu

Jie Liu*
University of Michigan
jiezzliu@umich.edu

Yuxuan Zhu
University of Michigan
yuxuanzh@umich.edu

Cong Yan
Microsoft Research
cong.yan@microsoft.com

Barzan Mozafari
University of Michigan
mozafari@umich.edu

Xinyu Wang
University of Michigan
xwangsd@umich.edu

ABSTRACT

Query rewriting is often a prerequisite for effective query optimization, particularly for poorly-written queries. Prior work on query rewriting has relied on a set of "rules" based on syntactic pattern-matching. Whether relying on manual rules or auto-generated ones, rule-based query rewriters are inherently limited in their ability to handle new query patterns. Their success is limited by the quality and quantity of the rules provided to them.

To our knowledge, we present the first *synthesis-based* query rewriting technique, SlabCity, capable of *whole-query optimization* without relying on any rewrite rules. SlabCity directly searches the space of SQL queries using a novel query synthesis algorithm that leverages a new concept called *query dataflows*. We evaluate SlabCity on four workloads, including a newly curated benchmark with more than 1000 real-life queries. We show that not only can SlabCity optimize more queries than state-of-the-art query rewriting techniques, but interestingly, it also leads to queries that are significantly faster than those generated by rule-based systems.

PVLDB Reference Format:
Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, Xinyu Wang. SlabCity: Whole-Query Optimization using Program Synthesis. PVLDB, 16(11): 3151 - 3164, 2023.
doi:10.14778/3611479.3611515

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/eidos06/SlabCity.

1 INTRODUCTION

Poorly-written database queries are a major problem in the industry [37, 51, 53–55, 62, 71]. Whether generated automatically by software (e.g., database-backed web apps [68]) or written manually by less experienced users (e.g., self-service BI users [1, 3, 6]), poorly-written queries significantly hinder the effectiveness of typical query optimizations performed by database systems [22, 68], whereby an optimal query plan is chosen for the given query. Query

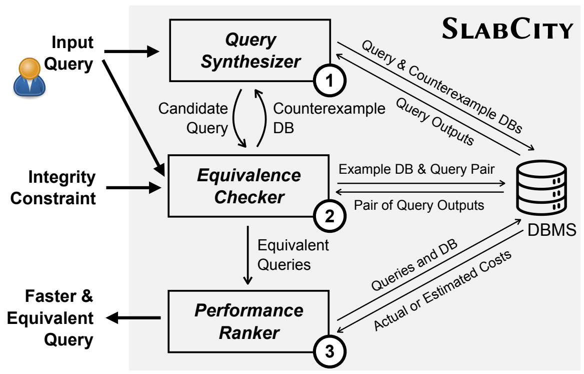


Figure 1: Schematic workflow of SlabCity.

rewriting — which transforms a query into another that is semantically equivalent but faster [31–33] — is thus a critical step in facilitating effective query optimization [23, 31–33, 41, 45, 46, 67, 68, 72].

Rule-based query rewriting. Query rewriting typically relies on a set of "rewrite rules" — crafted manually by experts or discovered automatically by tools — which essentially define an equivalence relation between queries. Specifically, given a query Q and a rewrite rule, if the pattern expressed in the rule matches Q, the rule would modify Q by replacing the matched part with a counterpart, generating a semantically equivalent query Q' that is likely to run faster. Examples of recent rule-based query rewriting techniques include WeTune [68] and Apache Calcite [17].

Drawback of rule-based query rewriting. Unfortunately, the effectiveness of rule-based approaches[1] hinges on the quality and quantity of the rewrite rules provided to them [68]. Yet, curating a large collection of high-quality rules is a very tedious, error-prone and time-consuming process that also requires deep expertise [31–33, 41, 45, 46]. While automated rule discovery techniques might help lessen the manual burden [68], they are fundamentally limited to situations where the input query has to match one of the patterns captured by their rules. In other words — as we will also show in this paper later — state-of-the-art rule-based query rewriters (e.g., WeTune [68]) miss many optimization opportunities when faced with new query patterns they have not seen before. We also provide several motivating examples from both real-life and benchmark workloads in Section 2 that demonstrate these drawbacks.

Synthesis-based query rewriting. Motivated by the fundamentally incomplete nature of rewrite rules, in this paper, we propose

*Rui Dong and Jie Liu contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611515

[1] Rule-based query *rewriting* should not be confused with rule-based query *optimization*; the latter is simpler, and hence much more common in practice [30].

a new query rewriting approach that does not need rewrite rules. Instead, it uses *program synthesis* to *directly* search for equivalent and faster queries. To our best knowledge, there are only two prior works[2] that use program synthesis for source-to-source query rewriting [67, 72], but they still either require rewrite rules or only allow local changes. Specifically, FGH-rule [67] first uses a rule to match a recursion pattern from the input Datalog query Q and generate an output query template with an unknown expression H, and then uses program synthesis to generate H such that the completed output query is equivalent to Q. The rule defines the search space for synthesis, and hence the overall effectiveness of this approach relies on the rule. The second work, Sia [72], uses synthesis to add additional predicates to the query, but is restricted to making only local changes in the **WHERE** clause and cannot optimize or restructure the rest of the query. To the best of our knowledge, we are the first to develop a synthesis-based query rewriting technique that (i) is capable of *whole-query* optimization and (ii) does not require any rewrite rules. We therefore call our technique, SlabCity.[3]

High-level workflow of SlabCity. As schematically shown in Figure 1, SlabCity takes a query Q_{in} and an integrity constraint ϕ; it returns a *semantically equivalent* query Q that is likely to run faster. SlabCity is composed of three key components: ① Query Synthesizer, ② Equivalence Checker, and ③ Performance Ranker. Here, ① and ② form a counterexample-guided inductive synthesis (CEGIS) loop [58]: the query synthesizer ① proposes a candidate query that is guaranteed to produce the same outputs as Q_{in} on a set of counterexample DBs, and the equivalence checker ② validates its semantic equivalence to Q_{in} against all inputs.

The equivalence checker uses three types of techniques, in their increasing order of overhead: a tester, a bounded verifier, and a full verifier (see Section 4.4 for the difference). If any of them finds a counterexample DB for a candidate query Q, then Q is rejected and the DB is given as feedback to the synthesizer. If the full verifier can prove equivalence (within a user-set timeout), then Q is marked "fully verified"; otherwise, Q would receive a "bounded verification" flag indicating it is correct only in a bounded sense (i.e., not fully). Equivalent queries with both types of flags will be ranked by the performance ranker ③ based on Q's latency (estimated by `EXPLAIN` or actual execution on a DBMS). At the end, if the (fastest) returned query has a "bounded verification" flag, users can manually inspect it to ensure its correctness. If it is "fully verified", no manual check is needed and the query can be safely used immediately. The tester, bounded verifier, and manual inspection are safeguards to address the limitations of full verification which is applicable to a subset of all rewrites (about 17%). We discuss the implications of this in detail in Section 5.5.

Target workload and use case. Our motivating use case is any scenario where either (1) the same query is written once but rerun many times, such as BI dashboards or database-backed web apps, or (2) the query is run once but its latency is much longer than several seconds (e.g., 5 seconds), such as ad-hoc or OLAP queries against big data. While SlabCity's time budget can be specified by users, in this work we use 5 seconds (that is, we have up to 5 seconds to rewrite a query into a faster one). This search cost is well justified and negligible in the aforementioned situations. For example, a developer creates a BI dashboard (and the queries therein) once, which is then used hundreds of times a day across the entire organization by various business users. Likewise, developers hardcode their queries in their web apps which are then invoked thousands of times as visitors interact with the website. In these scenarios, developers can invoke SlabCity as a final step to identify any optimizations before deploying their dashboard or web apps. Similarly, queries in modern data warehouses against terabytes of data will often take minutes and therefore spending an additional 5 seconds upfront to check if it can be rewritten into a more optimized form will be worthwhile even if the query will only run once. In terms of the query dialect, SlabCity supports an expressive subset of SQL, including nested queries and arbitrary join patterns (see the formal language definition in Section 3).

Challenges. However, SlabCity's key advantage of not relying on rules comes with three major challenges. First, given the expressiveness of our SQL language, searching for an equivalent query is a non-trivial task. A brute-force approach (e.g., enumerating programs in the order of program size) from the program synthesis literature [36, 65] would not scale. Second, during synthesis, non-equivalent queries need to be disproved by a verifier which typically involves Satisfiability Modulo Theory (SMT) solving and thus is in general costly. Finally, it is insufficient to find *any* equivalent query; we have to find one that is both equivalent and faster. This makes our problem even more challenging — we may need to find multiple equivalent queries in order to select a faster one.

Intuition. To address these challenges, our first key insight is that a query's dataflow — i.e., information that flows through different query operations — can be used to prioritize query search. In other words, dataflows of the input query could serve as a hint to help significantly reduce the number of queries to be enumerated. The intuition is that if a query Q is equivalent to Q_{in}, Q typically exhibits dataflows that are also manifested in Q_{in}. For example, if Q filters an aggregated column (e.g., using **HAVING SUM**), the same computations (e.g., calculating **SUM**) likely would also show up in Q_{in}. On the other hand, a query Q' that does *not* involve such dataflows is less likely to be equivalent and therefore can be de-prioritized. Our second insight is that we can leverage a lightweight testing approach to significantly lower the frequency of invoking a costly equivalence verifier. These insights combined allow us to develop a new CEGIS-based technique: the checker runs a tester before invoking an SMT-based verifier, and the query synthesizer utilizes dataflows to speed up the search. In particular, our synthesis technique stratifies the search space using a novel dataflow-based scoring function; we will expand on the technical details in Section 4.2.

Contributions. In summary, we make the following contributions.

- We propose the first synthesis-based query rewriting technique capable of whole-query optimization without requiring predefined rewrite rules.
- To the best of our knowledge, we are the first to define dataflows for SQL queries and exploit them for efficient query synthesis.
- We contribute a new benchmark to facilitate research on query rewriting research by curating more than 1000 real-life queries from LeetCode participants.

[2]See Section 6 for more applications of synthesis in the databases literature.
[3]Slab City is a spot in California known as the last free place in America, with no rules.

Table 1: **Q_1 is human-written. Q_2 is generated by SlabCity.**

Q_1	**SELECT DISTINCT** s1.gender, s1.day, **SUM** (s2.score) **FROM** scores **AS** s1 **JOIN** scores **AS** s2 **ON** s1.gender = s2.gender **WHERE** s2.day <= s1.day **GROUP BY** s1.gender, s1.day
Q_2	**SELECT** gender, day, **SUM** (score) **OVER** (**PARTITION BY** gender **ORDER BY** day) **FROM** scores

- Our comprehensive evaluation on a wide range of workloads and databases shows that, SlabCity can not only rewrite 7–68% more queries than state-of-the-art query rewriters, but also generate queries that are significantly faster (up to 4 orders of magnitude).

2 MOTIVATING EXAMPLES

In this section, we present a few examples to highlight the inherent limitations of rule-based query rewriting techniques and motivate the need for a synthesis-based whole-query optimization approach.

All examples in this section are from two workloads: (1) real-life queries written by LeetCode users, and (2) benchmark queries from the Apache Calcite project [12]. We later present more comprehensive experiments on both workloads among others in Section 5.

Example 1. Optimization with window functions. Consider LeetCode problem #1308: a competition is held among teams of different genders. Given the table called `scores` with three columns `(gender,day,score)` where `(gender,day)` is the primary key, the goal is to find the running total score for each gender on each day. Q_1 in Table 1 is the human-written solution for this problem (submitted by LeetCode participants) while Q_2 is SlabCity's automatically generated query after rewriting Q_1. Q_2 is more than 2000x faster than Q_1 on a database randomly populated with 1 million rows.

Q_1 is slow for two reasons. First, it uses a self-join to compute the running total, which generates a massive intermediate result. The same running total can be obtained using a window function as shown in Q_2. Second, the use of **DISTINCT** in Q_1 is redundant, since `(gender,day)` is the primary key.

It is difficult to express any local rewrite rule to do this kind of optimization for Q_1 because nearly every part of the query would have to change. Moreover, anticipating these kinds of patterns in advance and writing a rule for each would be tedious (if not impossible). Even if one can create rules for a query like Q_1, many constraints would need to be met to ensure correctness, necessitating a potentially very complex pattern-matching. However, SlabCity's synthesis-based optimization can deal with such cases easily because, instead of doing constraint checking and pattern-matching, SlabCity *directly* searches the query language (as defined in Figure 2), allowing it to discover semantically equivalent and faster queries that may be quite different *syntactically*. For these reasons, none of the state-of-the-art rule-based rewriters, such as `LearnedRewrite` (`LR`) [76] and WeTune (`WT`) [68], were able to optimize Q_1.[4]

Example 2. Exploiting integrity constraints. The following is LeetCode problem #1821. Given table `customers` with columns `(cid,year,revenue)` and `(cid,year)` as its primary key, the goal is to report customers with positive revenue in the year 2021. Q_3

[4]We explain how these state-of-the-art query rewriters work in more detail and report comprehensive experimental results in Section 5.

Table 2: **Q_3 is human-written. Q_4' and Q_4 are discovered by SlabCity (Q_4 is the final output as its faster than Q_4').**

Q_3	**SELECT** cid **FROM** (**SELECT** cid, **SUM** (revenue) **OVER** (**PARTITION BY** cid, year) **AS** r **FROM** customers **WHERE** year = 2021) **AS** tmp **WHERE** r > 0
Q_4'	**SELECT** cid **FROM** customers **WHERE** year = 2021 **GROUP BY** cid **HAVING SUM** (revenue) > 0
Q_4	**SELECT** cid **FROM** customers **WHERE** year = 2021 **AND** revenue > 0

from Table 2 is the human-written solution[5] for this task, while Q_4' and Q_4 are SlabCity's automatically generated queries for Q_3, which are 1.27x and 5.47x faster respectively (on a database with 1M rows). SlabCity can generate both Q_4' and Q_4 but returns the latter due to its superior performance.

Q_3 is slow because it uses an unnecessary window function with **PARTITION BY**. Q_4' achieves the same goal faster by replacing the window function with aggregation and **GROUP BY**. This is because `(cid,year)` is the primary key — once `year` is fixed, there is only a tuple for each unique value of `cid`. That is, partitioning by `(cid,year)` and grouping by `cid` when `year=2021` leads to the same groups. Exploiting this integrity constraint further, SlabCity is able to find an even more optimized query Q_4. This is because each `(cid,year)` will also determine a unique `revenue`, making the summation unnecessary. However, none of the state-of-the-art rule-based techniques could optimize or even rewrite Q_3 to anything. Capturing this kind of rewrite for rule-based techniques is nearly impossible, as they heavily rely on pattern-matching and their ability to exploit integrity constraints is largely limited to local changes (e.g., removing redundant grouping columns or redundant left joins). In contrast, because SlabCity is not restricted to rewrite rules fundamentally, as long as there is a better query in the query space, it will eventually find it.

Example 3. Eliminating redundant joins. Calcite comes with a rich set of test cases to check if its rewrite rules function properly. Q_5 in Table 3 is one of those test cases ("testPushAggregateThroughOuterJoin14") where Calcite rewrites Q_5 to Q_6. Here, `emp` is a table with three columns `(empno,ename,mgr)` with `empno` as its primary key. Q_5 performs a full self-join of `emp` on the `mgr` column, grouped by the join key. This is essentially equivalent to using **DISTINCT** to return only distinct values. Q_6 is the optimized query using Calcite rules, which is 926x faster than Q_5 on a database with 4M rows. Q_7 is the query automatically generated by SlabCity, which is 1833x faster than Q_5 on the same database.

Q_6 is significantly faster than Q_5 because pushing **GROUP BY** before the self-join leads to significantly smaller join operands and thus a more efficient execution; however, this is still not the best rewrite possible. Interestingly, self-join can be eliminated altogether in this case, leading to an even more significant speed-up, which is exactly what SlabCity does by rewriting Q_5 into Q_7. In fact, this is

[5]While a database expert might be surprised why the user missed the integrity constraint and wrote such an inefficient query in the first place, situations like this are quite common in the industry for two reasons: (1) most queries are rewritten by users who are not SQL proficient, such as business users and financial analysts [51, 62] , and (2) modern warehouses have hundreds of tables, and the knowledge of the schema and integrity constraints are thus scattered across different teams, especially in larger organizations [16, 59].

Table 3: Q_5 is a test query from Calcite, Q_6 is the rewritten version of Q_5 using Calcite rules, and Q_7 is SlabCity's output.

Q_5	`SELECT e0.mgr AS mgr0, e1.mgr AS mgr1`
	`FROM emp AS e0 FULL JOIN emp AS e1 ON e0.mgr = e1.mgr`
	`GROUP BY e0.mgr, e1.mgr`

Q_6	`SELECT e0.mgr AS mgr0, e1.mgr AS mgr1`
	`FROM (SELECT mgr FROM emp GROUP BY mgr) AS e0`
	`    FULL JOIN (SELECT mgr FROM emp GROUP BY mgr) AS e1`
	`    ON e0.mgr = e1.mgr`
	`GROUP BY e0.mgr, e1.mgr`

Q_7	`SELECT mgr AS mgr0, mgr AS mgr1`
	`FROM emp`
	`GROUP BY mgr`

Table 4: Q_8 is a test query from Calcite, Q_9 is the rewrite using Calcite rules, and Q_{10} is SlabCity's output.

Q_8	`SELECT ename, MIN (empno) AS e FROM (`
	`    SELECT * FROM emp`
	`    UNION ALL`
	`    SELECT * FROM emp`
	`) AS t GROUP BY ename`

Q_9	`SELECT ename, MIN (e) AS e FROM (`
	`    SELECT ename, MIN (empno) AS e FROM emp GROUP BY ename`
	`    UNION ALL`
	`    SELECT ename, MIN (empno) AS e FROM emp GROUP BY ename`
	`) AS t GROUP BY ename`

Q_{10}	`SELECT ename, MIN (e) AS e`
	`FROM (SELECT ename, MIN (empno) AS e`
	`        FROM emp`
	`        GROUP BY ename) AS t`
	`GROUP BY ename`

not limited to Calcite: none of the state-of-the-art rule-based query rewriters found this opportunity,[6] simply because they fail to capture the information that the joined columns may come from the same table and hence be identical — a fact that could be exploited to eliminate redundancy. Similar to Examples 1 and 2, it is tedious, if not impossible, to create rewrite rules for this type of optimization. In particular, identifying self-joins requires semantic information that the two joined operands are essentially the same relation. Even if one wrote a very specific rule for

```
FROM T AS T1 JOIN T AS T2
```

to find self-joins syntactically, they would still miss out on queries expressed as

```
FROM T AS T1 JOIN (SELECT * from T) AS T2
```

In contrast, because of using query synthesis, Q_7 is naturally within SlabCity's search space and SlabCity can successfully find it as a query with minimal redundancy and best performance.

Example 4. Optimizating set operations. Q_8 in Table 4 is yet another test case ("testPushMinThroughUnion") from Calcite, which is rewritten to Q_9 using Calcite rules, slightly improving its latency (±2%) by pushing `GROUP BY` past `UNION ALL`. On the other hand, SlabCity rewrites Q_9 to Q_{10}, which is 1.7x faster.

Similar to Example 3, state-of-the-art rule-based rewriters fail to exploit the fact that both sides of `UNION ALL` are identical, and thus fail to eliminate the redundant computation. In fact, neither `LearnedRewrite` nor WeTune were able to optimize Q_8 (the former could rewrite it to Q_9 whereas the latter was not even able to rewrite it to anything at all). Again, because SlabCity uses query synthesis, it is able to find Q_{10} which is both equivalent and faster.

Discussion. In our evaluation, we have come across many other interesting examples where state-of-the-art query rewriting techniques were not able to optimize the input query or even rewrite it at all, while SlabCity could (see Section 5.2). Even when they were able to optimize the query, SlabCity's output oftentimes was significantly faster (see Section 5.3). Note that in this paper, we differentiate two terms *optimize* vs. *rewrite*: the former means the technique can rewrite the input query to an output query that is faster, whereas the latter means the technique can rewrite to some query which may or may not be faster. In other words, optimizing is harder than rewriting. Our intention of sharing these motivating examples in Section 2 is to demonstrate why it is inherently difficult and error-prone to create enough rules that can handle complex and unseen situations. In contrast, a synthesis-based approach does not need a supply of hard-coded rules and can discover optimization opportunities by searching the SQL language *directly.*

3 PROBLEM SETUP

This section presents the query language and integrity constraints considered by SlabCity, followed by our problem definition.

Query space. See Figure 2 for the formal language that supports a wide subset of SQL such as arbitrary nesting and join patterns.

Integrity constraints. Below are the types of integrity constraints SlabCity currently supports:

- Primary and foreign keys.
- Comparisons within a row — e.g., `T.StartTime` < `T.EndTime`.
- Implication constraints within a row. For example, `T.EmailType` ≠ *spam* ⟶ `T.action` = `NULL`.
- Whether or not a column can be `NULL`.
- Range constraints (for columns of integer or numeric types).
- Enum types (e.g., column `Device` can draw from { *S8*, *iPhone* }).

Problem statement. Now we are ready to define our problem.

Definition 3.1. Given a query Q_{in} and an integrity constraint ϕ, find a query Q from our query language, such that the following two conditions hold:

(1) Q is semantically equivalent to Q_{in} with respect to ϕ. That is, Q and Q_{in} produce the same output on any database D that meets the integrity constraint ϕ.
(2) Q runs faster than Q_{in}. Here, query performance is measured using `EXPLAIN` or the actual execution of the query.

4 SYNTHESIS-AIDED QUERY OPTIMIZATION

4.1 Top-Level Algorithm

Our top-level algorithm is shown in Algorithm 1. It accepts as input a query Q_{in} and an integrity constraint ϕ, and returns an output query Q that is semantically equivalent to and faster than Q_{in}. Q is $\perp$ if it does not find one such query within the given time limit.

Let us explain Algorithm 1 in more detail. At a high-level, our algorithm is a *new instantiation* of the counterexample-guided inductive synthesis (CEGIS) paradigm [58] — which has found great success in the programming languages community — for our query

[6] Similar to Calcite, LearnedRewrite [76] was only able to rewrite Q_5 into Q_6 but did not eliminate the self-join. WeTune was not able to rewrite Q_5 at all.

```
Query        Q    ::= an input table
                    | SELECT L FROM Q WHERE ψ
                    | SELECT L FROM Q GROUP BY cols HAVING ψ
                    | SELECT L FROM Q ORDER BY cols
                    | Q INNER JOIN Q ON ψ
                    | Q LEFT JOIN Q ON ψ
Target List  L    ::= [t AS alias, ··· , t AS alias]
                    | DISTINCT [t AS alias, ··· , t AS alias]
Target       t    ::= col | α(col) | α(DISTINCT col)
                    | α(col) OVER(PARTITION BY cols ORDER BY cols)
                    | ω OVER(PARTITION BY cols ORDER BY cols)
Column List  cols ::= [col, ··· , col]
Column       col  ::= alias | a column from an input table
Condition    ψ    ::= E op E | ψ ∧ ψ | ψ ∨ ψ
Expression   E    ::= col | α(col) | α(DISTINCT col) | const
Agg. Func.   α    ::= MAX | MIN | AVG | SUM | COUNT
Window Func. ω    ::= DENSE_RANK | RANK
```

Figure 2: Syntax of our query language.

optimization problem. Our approach consists of (a) an inductive synthesizer (i.e., our query synthesizer) that aims to find a query Q which satisfies a set $\mathcal{E}$ of *counterexamples* and (b) a query checker (i.e., our equivalence checker) which checks whether or not Q is equivalent to Q_{in} and generates a counterexample E if not. One key advantage of using CEGIS is to enable efficient checking: checking a candidate query against $\mathcal{E}$ (line 4) is typically significantly cheaper than calling an equivalence checker (line 5, e.g., SPES [74]). In other words, verification is used parsimoniously only when necessary. In our context, a counterexample E is an input DB together with the desired output table returned by Q_{in}.

In addition to using counterexamples, SlabCity also uses a novel *score* function that defines dataflows for queries in order to guide the query synthesis process. While we defer the formal definition of *score* to Section 4.2, from a high-level, it assigns a non-positive integer score[7] to each query Q such that, Q with a higher score is more likely to be an optimized query (i.e., semantically equivalent to and faster than Q_{in}) and vice versa. Initially, $\mathcal{E}$ has no counterexamples and $\widetilde{Q}$ is a empty set that stores equivalent queries (line 1). Then, we enter a CEGIS loop (lines 2–7). At line 3, it *lazily* enumerates all queries in non-ascending order of their scores. In other words, SlabCity prioritizes the search for candidate queries Q that are likely to optimize Q_{in} (i.e., with a higher score). Line 4 checks if Q meets $\mathcal{E}$: if not, it continues to the next candidate. If Q passes $\mathcal{E}$, we invoke the equivalence checker (line 5) to see if Q is equivalent — it returns a new counterexample E if not equivalent, or returns *null* otherwise. For the latter case, line 6 adds Q to $\widetilde{Q}$; for the former, line 7 adds E to $\mathcal{E}$. Upon termination, we rank queries in $\widetilde{Q}$ (line 8) and return a "best" query that we believe can *optimize* Q_{in}.

Example 4.1. Suppose Q_{in} is query Q_1 from Table 1. Our CEGIS-based algorithm will begin with query candidates with score 0: Q' below is one such query. Q' is obviously not equivalent to Q_{in}, and our equivalence checker gives a counterexample E below. Note that E consists of an input DB D and the output that Q_{in} returns on D. A query eventually returned by our tool will pass this counterexample.

[7]While *non-positive* scores may seem counter-intuitive, they can be viewed as the negation of the cost, i.e., the higher the score, the lower the cost.

Algorithm 1 Top-level algorithm of SlabCity.

```
procedure Optimize ($Q_{in}$, $\phi$)
input: An input query $Q_{in}$ and an integrity constraint $\phi$.
output: An equivalent and faster output query $Q$.
1: $\mathcal{E} := \emptyset$; $\widetilde{Q} := \emptyset$;
2: while not timed out do
3:     for all $Q \in$ LazyEnumerate($Q_{in}$) do          ▷ ① Query synthesizer.
4:         if $Q$ doesn't pass $\mathcal{E}$ then continue;    ▷ Check against counterexs.
5:         $E :=$ CheckEquivalence($Q, Q_{in}, \phi$);      ▷ ② Equivalence checker.
6:         if $E = null$ then $\widetilde{Q}.add(Q)$        ▷ null means $Q$ is equivalent to $Q_{in}$.
7:         else $\mathcal{E}.add(E)$;                   ▷ Otherwise, $E$ is a new counterexample.
8: $Q :=$ RankPerformance($\widetilde{Q}, Q_{in}$);         ▷ ③ Performance Ranker.
9: return $Q$;
```

A counterexample E consists of an input database D (left) and an output table of Q_{in} on D.

```
Q' : SELECT gender, day,
            SUM (score)
     FROM scores
     GROUP BY gender, day
```

```
Q'' : SELECT gender, day,
             SUM (score)
             OVER (PARTITION BY gender
                   ORDER BY day)
      FROM scores
      GROUP BY gender, day
```

gender	day	score
a	1	1
a	2	2

gender	day	score
a	1	1
a	2	3

On the other hand, the query Q'' next to Q above is equivalent to Q_{in}. However, Q'' is determined to be slower than Q_2 from Table 1 by our performance ranker. SlabCity was able to find both Q_2 and Q'' which are added to $\widetilde{Q}$; however, SlabCity returns Q_2 in the end, since it is ranked the highest.

4.2 Dataflow-Based Query Score Function

As we can see, a key challenge underlying the success of Algorithm 1 is how to design a good score function as well as how to develop an algorithm that can effectively use the score function to prioritize the search. This section first addresses the score function design.

Key idea: scoring queries based on dataflows. Recall that we use a score function to quantitatively rank queries in terms of their likelihood of optimizing a user-provided input query Q_{in}. That is, given Q_{in} and Q, a desired score function should assign Q with a higher score if Q is indeed semantically equivalent to and faster than Q_{in}. At the same time, it should minimize false positives: that is, it should not assign high scores to too many inequivalent queries.

To design such a score function, our key insight is that, a query Q that indeed optimizes Q_{in} typically exhibits *dataflows* that are also manifested in Q_{in}. For example, if Q filters an aggregated column (e.g., using **HAVING SUM**), the same computations (e.g., calculating **SUM**) likely would also show up in Q_{in}, although the computations might be organized *syntactically differently* (e.g., first calculating **SUM** in **SELECT** and then using **WHERE** to filter, which is slower). In other words, our dataflow-based score function is designed to favor queries Q that involve dataflows from Q_{in} — the rationale is that, since Q_{in} is functionally correct, its underlying computations are likely sufficient for generating an optimized query.

However, we do not require Q to use up all dataflows from Q_{in}. In other words, Q_{in} may exhibit redundant dataflows that are not necessary for computing the same result and thus can be optimized

$$\begin{aligned}\delta ::= &\ \textit{an input table} \mid \textit{a column from an input table}\\ &\mid \alpha(\delta,\cdots,\delta) \mid \omega \mid [\delta,\cdots,\delta] \mid op(\delta,\delta) \mid \delta \wedge \delta \mid \delta \vee \delta\\ &\mid \textbf{PARTITION BY}(\delta) \mid \textbf{ORDER BY}(\delta) \mid \textbf{GROUP BY}(\delta)\end{aligned}$$

Figure 3: Dataflow language.

away. For example, if Q_{in} involves a redundant **JOIN**, an optimized query Q may contain only dataflows from one branch of the **JOIN**.

Finally, if a query Q' contains dataflows that are not exhibited in Q_{in}, oftentimes Q' is not equivalent and hence we can assign it with a lower score. For example, suppose Q_{in} calculates **SUM** over a column. It is less likely that a query Q' can accomplish the same goal with only **AVG**. Similarly, if Q_{in} and Q' both perform filtering but on two different columns that are unrelated, it is more plausible to believe they are not semantically equivalent.

Note that, since our score is calculated based on semantic information (i.e., dataflows) instead of syntactic features, a syntactically simpler query (e.g., one with fewer lines) may receive a lower score than a more complex query. In other words, our algorithm first prioritizes generating queries with higher scores, and then prioritizes syntactically smaller queries when their scores are the same (see Section 4.3). This is an advantage since it will allow us to uncover non-trivial rewrites sooner if they are more promising.

Dataflows. Our observations suggest a score function design that takes two queries (i.e., Q_{in} to be optimized and Q to be scored), extracts dataflows for each, and calculates the score based on these dataflows. In what follows, we first formalize the notion of dataflow. Then, we present the dataflow extraction algorithm.

Definition 4.2 (Dataflow). A dataflow δ (defined in Figure 3) for a given query Q is a sequence of SQL operations that are performed during Q's execution on one or more input tables or their columns.

Let us explain our dataflow language more formally (see Figure 3). In the base cases, a dataflow δ is either an input table T or a column from an input table. In the recursive cases, δ captures operations performed on top of such base dataflows. For example, $\alpha(\delta,\cdots,\delta)$ describes the application of an aggregate function over dataflows for argument columns. $[\delta,\cdots,\delta]$ is the composition of columns to form a table (e.g., via **SELECT**). Expressions, such as = and ≤, involves comparison logics, which is what $op(\delta,\delta)$ is designed for. The language also includes boolean combinations of dataflows in order to represent dataflows from the filtering conditions in a query. Finally, we capture other SQL operations, such as **PARTITION BY**, **ORDER BY**, **GROUP BY**, etc.

Example 4.3. In this example, we show some sample dataflows for Q_{in} from Example 4.1:

{`scores.score`, `scores.day` ≤ `scores.day`, **SUM**(`scores.score`), $\cdots$ }

We have `scores.score` because Q_{in} uses data from `score` column in `scores` table. We have **SUM**(`scores.score`), since Q_{in} performs summation on this column. Similarly, `scores.day` ≤ `scores.day` because an intermediate step of Q_{in} does a comparison between these two columns.

Extracting dataflows from queries. Now let us talk about how to extract a set Δ of dataflows from a query Q. Figure 4 shows how to extract dataflows where we use a judgment of the form:

$$Q_{ctx} \vdash \textsc{AllDfs}(P) : \Delta$$

This means: given a "context" Q_{ctx} associated with P, Δ is the set of *all* dataflows that are manifested in P. Here, P may be a query, a condition, a target list, etc. In general, P is associated with a context Q_{ctx} — e.g., if P is a target list, Q_{ctx} is the query that produces the table that the targets in P correspond to. Having a context allows us to trace the flow of data to the original input tables.

Let us first explain how to extract dataflows for a query Q. The first rule in Figure 4 concerns the base case where Q is an input table T: in this case, the result is a singleton set with T. Intuitively, this means, the computation in Q involves only T and nothing else. The next one concerns selection — it states that, the dataflow set Δ of a **SELECT** query is the union of three sets: Δ_1 for the query Q to select from, Δ_2 for the filtering condition ψ, and Δ_3 for the target list L. Conceptually, Δ_1 contains dataflows manifested in *all components* of Q; that is, if Q is a nested query, Δ_1 would also include dataflows from the sub-queries. Similarly, Δ_2 and Δ_3 capture computations performed in ψ and L. (3) is very similar to (2) in that it also takes the union of dataflow sets for each of the arguments of **JOIN**. The join logic is implicitly captured in Δ_3 for the join condition ψ.

Next, let us examine the extraction rules for a condition ψ, which can be used as a join condition or used in **WHERE** or **HAVING**. Looking at (4), for a boolean combination *lop* (e.g., ∨) of multiple conditions, we would first recursively invoke AllDfs to obtain *all dataflows* Δ_i for each ψ_i. In addition, since *lop* itself performs a logical operation, the final dataflow set also includes $lop(\delta_1,\delta_2)$: here, δ_i is the dataflow for ψ_i that (different from Δ_i) does *not* include dataflows from ψ_i's components, and we use a separate function Df (different from AllDfs) to compute δ_i. The key difference between AllDfs and Df is that, the former includes all dataflows for all components in P, whereas the latter only concerns one dataflow for P itself. The next rule (5) takes care of the base case where the condition ψ is an application of *op* (e.g., =) over expressions (e.g., columns). It also makes use of Df to retrieve the dataflow for an expression E_i.

Let us dive a little deeper to see how Df works. (6) states that the dataflow of a logical operation (e.g., ∧) is composed of dataflows of the two arguments ψ_1, ψ_2. (7) is very similar except that it concerns comparison operations (like =) and it invokes Df on expressions E_i such as a column, an aggregate function applied to a column, or a constant. (8)-(13) detail how to extract dataflows for expressions. (8) says the dataflow for a constant is the constant itself. (9) states that, given an *input* table T, the dataflow for its column *col* is $T.col$. (10) considers extracting the dataflow for a column *col* — which may be a column alias — given a **JOIN** query: it recursively extracts the dataflow for *col* given Q_i that *col* comes from. (11) is similar: it first identifies the target t from the target list L that corresponds to *col* and then invokes Df on t given the query Q being selected from. Finally, (12) and (13) extract dataflows for aggregate functions.

Example 4.4. Let us explain how to extract the dataflows for the following component (i.e., a comparison) in Q_1 from Table 1.

```
s2.day <= s1.day
```

The dataflow set of this expression is the union of three sets of dataflows: (1) the dataflows of its left argument ({`scores.day`}), (2) the dataflows of the right argument (namely {`scores.day`}), and (3) the dataflow of the comparison (i.e., {`scores.day` ≤ `scores.day`}). This essentially means that the query uses data from `day` column of

$\boxed{\textsc{AllDfs}\text{ algorithm that extracts all dataflows for queries}}$

$$(1)\ \frac{T \text{ is an input table}}{\vdash \textsc{AllDfs}(T) : \{T\}} \quad (2)\ \frac{\vdash \textsc{AllDfs}(Q) : \Delta_1 \quad Q \vdash \textsc{AllDfs}(\psi) : \Delta_2 \quad Q \vdash \textsc{AllDfs}(L) : \Delta_3}{\vdash \textsc{AllDfs}(\mathtt{SELECT}\ L\ \mathtt{FROM}\ Q\ \mathtt{WHERE}\ \psi) : \Delta_1 \cup \Delta_2 \cup \Delta_3} \quad (3)\ \frac{\vdash \textsc{AllDfs}(Q_1) : \Delta_1 \quad \vdash \textsc{AllDfs}(Q_2) : \Delta_2 \quad Q_1\ \mathtt{JOIN}\ Q_2 \vdash \textsc{AllDfs}(\psi) : \Delta_3}{\vdash \textsc{AllDfs}(Q_1\ \mathtt{JOIN}\ Q_2\ \mathtt{ON}\ \psi) : \Delta_1 \cup \Delta_2 \cup \Delta_3}$$

$\boxed{\textsc{AllDfs}\text{ algorithm that extracts all dataflows for conditions}}$

$$(4)\ \frac{Q \vdash \textsc{AllDfs}(\psi_i) : \Delta_i \quad Q \vdash \textsc{Df}(\psi_i) : \delta_i}{Q \vdash \textsc{AllDfs}(\psi_1\ lop\ \psi_2) : \Delta_1 \cup \Delta_2 \cup \{lop(\delta_1, \delta_2)\}} \quad (5)\ \frac{Q \vdash \textsc{AllDfs}(E_i) : \Delta_i \quad Q \vdash \textsc{Df}(E_i) : \delta_i}{Q \vdash \textsc{AllDfs}(E_1\ op\ E_2) : \Delta_1 \cup \Delta_2 \cup \{op(\delta_1, \delta_2)\}}$$

$\boxed{\textsc{Df}\text{ algorithm that extracts dataflows for conditions, expressions and targets}}$

$$(6)\ \frac{Q \vdash \textsc{Df}(\psi_i) : \delta_i}{Q \vdash \textsc{Df}(\psi_1\ lop\ \psi_2) : lop(\delta_1, \delta_2)} \quad (7)\ \frac{Q \vdash \textsc{Df}(E_i) : \delta_i}{Q \vdash \textsc{Df}(E_1\ op\ E_2) : op(\delta_1, \delta_2)} \quad (8)\ \frac{}{Q \vdash \textsc{Df}(const) : const} \quad (9)\ \frac{T \text{ is an input table}}{T \vdash \textsc{Df}(col) : T.col} \quad (10)\ \frac{col \in \textsc{Columns}(Q_i) \quad Q_i \vdash \textsc{Df}(col) : \delta_i}{Q_1\ \mathtt{JOIN}\ Q_2\ \mathtt{ON}\ \psi \vdash \textsc{Df}(col) : \delta_i}$$

$$(11)\ \frac{(t\ \mathtt{AS}\ col) \in L \quad Q \vdash \textsc{Df}(t) : \delta}{\mathtt{SELECT}\ L\ \mathtt{FROM}\ Q\ \mathtt{WHERE}\ \psi \vdash \textsc{Df}(col) : \delta} \quad (12)\ \frac{Q \vdash \textsc{Df}(col) : \delta}{Q \vdash \textsc{Df}(\alpha(col)) : \alpha(\delta)} \quad (13)\ \frac{Q \vdash \textsc{Df}(col) : \delta}{Q \vdash \textsc{Df}\big(\alpha(col)\ \mathtt{OVER}(\mathtt{PARTITION\ BY}\ cols_1\ \mathtt{ORDER\ BY}\ cols_2)\big) : \alpha(\delta)}$$

Figure 4: Inference rules that explain how dataflow extraction works for some key constructs in our query language.

the input `scores` table and performs a comparison where the left and right arguments come from the `day` column of `scores` table as well. This provides clues to guide the search.

Dataflow-based scoring. Now we define our *score* function, which scores a program P (potentially associated with a context Q_{ctx}) with respect to an input query Q_{in}. First, it extracts two sets of dataflows, i.e., Δ_{in} and Δ, for Q_{in} and P respectively using the algorithm from Figure 4. Then, it computes the set difference, i.e., $\Delta_{diff} = \Delta \setminus \Delta_{in}$, which gives the dataflows in P but not in Q_{in}. Finally, we define $score(P|Q_{ctx}, Q_{in}) = -|\Delta_{diff}|$. As we can see, 0 is the highest possible score: in this case, all of P's dataflows are from Q_{in}. For brevity, we use notation $score(P|Q_{ctx})$ when Q_{in} is clear from the context.

Monotonicity. An important property of our *score* function is that it is *monotonic*. That is, given Q_{in} and Q_{ctx}, for any component P' of P (e.g., P' is a sub-query of query P), $score(P'|Q_{ctx}) \geq score(P|Q_{ctx})$. The proof is obvious due to the monotonic nature of our dataflow extraction algorithm and our *score* function: the dataflow set for P' is always a subset of that for P, hence P's score is no less than P's. The implication is: when composing multiple programs $P_1, \cdots, P_n$ to form a bigger program P, P's score is never higher than the score of any P_i's. The next Section 4.3 presents an algorithm that uses this property to effectively guide the query synthesis process.

4.3 Prioritizing Search using Score Function

Our key idea is to leverage the *monotonicity* property of our score function to develop a *stratified search algorithm* that enumerates programs in layers: it first finds all queries with score 0 (i.e., the highest possible score) *without* considering those with lower scores, then generates all queries with score -1 (i.e., the second highest) *again without* constructing those with lower scores, and so on.

In particular, to generate *all* queries $\widetilde{Q}_S$ whose score is exactly S, we track only those queries $\widetilde{Q}_{\geq S}$ whose score is *at least* S, because according to the monotonicity property, $\widetilde{Q}_{\geq S}$ is sufficient to construct $\widetilde{Q}_S$. This allows for a dynamic programming design that lazily enumerates queries, as presented in Algorithm 2. It takes the input query Q_{in} and returns a stream of queries in a non-ascending order of their scores. Line 1 initializes $\widetilde{Q}_{\geq 1}$ to empty set since the highest possible score is 0. Then, the loop at lines 2–8 populates

Algorithm 2 LazyEnumerate search algorithm.

```
procedure LazyEnumerate ($Q_{in}$)
input: Input query $Q_{in}$.
output: A stream of candidate queries in non-ascending order of scores.
1: $\widetilde{Q}_{\geq 1} := \emptyset$;                    ▷ All queries scored at least 1, which is empty.
2: for $S = 0, -1, -2, \cdots$ do
3:     $\widetilde{Q}_{\geq S} := \widetilde{Q}_{\geq S+1}$;            ▷ All queries scored $\geq S$, initially set to $\widetilde{Q}_{\geq S+1}$.
4:     while true do
5:         $\widetilde{Q}'_S := \textsc{NewQueriesWithScoreS}(Q_{in}, \widetilde{Q}_{\geq S}, S)$;
6:         if $\widetilde{Q}'_S = \emptyset$ then break;            ▷ Exit if no more new queries scored $S$.
7:         yield $\widetilde{Q}'_S$;
8:         $\widetilde{Q}_{\geq S} := \widetilde{Q}_{\geq S} \cup \widetilde{Q}'_S$;          ▷ All queries scored at least $S$ so far.
```

$$(1)\ \frac{T \text{ is input table} \quad score(T) = S}{T \in \widetilde{Q}'_S} \quad (2)\ \frac{Q_i \in \widetilde{Q}_{\geq S} \quad \psi \in \widetilde{\psi}_{\geq S}(Q) \quad score(Q') = S}{(Q' = Q_1\ \mathtt{JOIN}\ Q_2\ \mathtt{ON}\ \psi) \in \widetilde{Q}'_{\geq S}}$$

$$(3)\ \frac{\psi_i \in \widetilde{\psi}_{\geq S}(Q) \quad score(\psi'|Q) \geq S}{(\psi' = \psi_1\ lop\ \psi_2) \in \widetilde{\psi}_{\geq S}(Q)} \quad (4)\ \frac{E_i \in \widetilde{E}_{\geq S}(Q) \quad score(\psi'|Q) \geq S}{(\psi' = E_1\ op\ E_2) \in \widetilde{\psi}_{\geq S}(Q)}$$

$$(5)\ \frac{const \text{ is used in } Q_{in}}{const \in \widetilde{E}_{\geq S}(Q)} \quad (6)\ \frac{col \in Cols(Q) \quad score(col|Q) \geq S}{col \in \widetilde{E}_{\geq S}(Q)}$$

$$(7)\ \frac{Q \in \widetilde{Q}_{\geq S} \quad L \in \widetilde{L}_{\geq S}(Q) \quad \psi \in \widetilde{\psi}_{\geq S}(Q) \quad score(Q') = S}{(Q' = \mathtt{SELECT}\ L\ \mathtt{FROM}\ Q\ \mathtt{WHERE}\ \psi) \in \widetilde{Q}'_S}$$

$$(8)\ \frac{t_i \in \widetilde{t}_{\geq S}(Q) \quad alias_i \text{ is a fresh column alias} \quad score(L|Q) \geq S}{(L = [t_1\ \mathtt{AS}\ alias_1, \cdots, t_n\ \mathtt{AS}\ alias_n]) \in \widetilde{L}_{\geq S}(Q)}$$

Figure 5: Inference rules for NewQueriesWithScoreS.

each $\widetilde{Q}_{\geq S}$ for all possible scores in descending order from 0. Specifically, given S, $\widetilde{Q}_{\geq S}$ is initialized to $\widetilde{Q}_{\geq S+1}$ (line 3). The inner loop (lines 4–8) iteratively constructs new queries $\widetilde{Q}'_S$ with score exactly S from $\widetilde{Q}_{\geq S}$ (line 5) until no more new queries are generated (line 6). Line 7 yields new queries $\widetilde{Q}'_S$ which are then added to $\widetilde{Q}_{\geq S}$ at line 8. Note that the while loop in Algorithm 2 may be non-terminating if the score function is based on *sets* of dataflows: it is possible to keep generating new queries with the same set of dataflows (hence the same score). To ensure termination, dataflows from a program are represented as a *multiset* where duplicate dataflows are counted multiple times; therefore, one cannot keep constructing new queries without decreasing the score.

Next, let us explain how NewQueriesWithScoreS (invoked at line 5 of Algorithm 2) works. Due to space limit, Figure 5 shows a key subset [8] of inference rules that describe how to construct a new query Q' with score S from an existing set $\widetilde{Q}_{\geq S}$ of queries with at least score S. Rule 1 is the base case stating that an input table T is generated if T has score S. Rule 2 describes a recursive case: we join Q_1 and Q_2 using condition ψ to generate a **JOIN** query Q' if Q_1 and Q_2 are both in $\widetilde{Q}_{\geq S}$, ψ has a score of at least S given context Q, and the score of Q' is exactly S. Rules 3–6 explain how conditions with a score of at least S can be generated, where Rules 5–6 provide the base cases and Rules 3–4 recursively build larger conditions. Rule 7 is an example to illustrate how to generate **SELECT** queries, which additionally involves generating target lists L with score at least S. Rule 8 explains how to generate such target lists. Note that, while implicit in the rule specification, we return Q' only if it is not already in $\widetilde{Q}_{\geq S}$.

In summary, new queries are generated in a bottom-up manner. That is, when synthesizing queries with score S, we only use queries with a score of at least S as building blocks. Further, among queries with the same score, we prioritize syntactically smaller queries.

4.4 Checking Query Equivalence

So far, we have seen how the inductive synthesizer in the CEGIS-based algorithm works. In this section, we explain how the equivalence checker works. As mentioned earlier, SlabCity incorporates a range of SQL query equivalence checking techniques, including empirical testing as well as formal verification. These techniques are complementary to each other: testing is relatively cheap but cannot prove equivalence, while verification can provide an equivalence guarantee but tends to be significantly more expensive.

Syntax-based query testing. To the best of our knowledge, there is little work on query equivalence testing for an expressive SQL language like ours. Thus, we develop a new tester specifically targeting our complex language. Given an input query Q_{in} and a candidate query Q, as well as an integrity constraint ϕ, the tester returns either (1) a database D which satisfies ϕ such that Q_{in} and Q produce different tables on D or (2) "unknown" indicating it is not able to find a counterexample. In the latter case, it is possible that Q_{in} and Q are indeed equivalent, or they are not equivalent but the tester is not able to disprove it. The key challenge is how to disprove many non-equivalent query pairs from an expressive language without being too costly. Randomly generating test inputs does not work.

Our key observation is that, we can extract hints from Q_{in} and Q using a mostly syntactic analysis to effectively guide the input generation process. Below are some of our key insights.

- *Leveraging filter conditions.* We observe that databases producing *non-empty* outputs tend to be useful for disproving equivalence. Therefore, we extract filter conditions (such as those in **WHERE**) from Q_{in} and populate the test database with rows that satisfy these conditions. We also collect the **JOIN** conditions to generate inputs that do not produce empty intermediate join results.
- *Duplicating non-key values.* A second source of non-equivalence that we have observed is from wrong logic around operations like addition/removal and from certain keywords such as **DISTINCT** being placed in a wrong location. Therefore, our tester identifies such operations and keywords to generate potentially useful test inputs accordingly. For example, if Q has **DISTINCT** on a column C, we will generate a test DB with duplicated values in C, with the goal of triggering the **DISTINCT** logic.
- *Duplicating* **GROUP BY** *columns.* Similar to the previous insight, we also observed a common class of incorrect query candidates with wrong **GROUP BY** clauses. While certain grouping strategies definitely lead to equivalent queries that are significantly faster, some of them produce non-equivalent ones. Therefore, our tester looks for columns in **GROUP BY** of Q that are different from those in Q_{in}. Then, it generates test DBs that contain duplicated values in such columns.

In general, each insight above leads to a set of test input DBs, all of which are encoded into a logical formula φ_1. We also encode the integrity constraint into a formula φ_2. We use a constraint solver (such as Microsoft Z3) to generate satisfying assignments for $\varphi_1 \wedge \varphi_2$, where each satisfying assignment corresponds to one DB instance. If φ_1 and φ_2 are contradictory (i.e., $\varphi_1 \wedge \varphi_2$ is unsatisfiable), it means no valid test inputs can be generated to differentiate Q_{in} and Q. In that case, we resort to subsequent verification approaches.

Bounded verification. In addition to testing, we also use a bounded verifier to check equivalence and find counterexamples: it considers all inputs in a bounded space and thus is more exhaustive than our tester. SlabCity incorporates existing bounded verifiers (i.e., those in Cosette [26, 66]). In particular, if verified, it guarantees that Q_{in} and Q produce the same output for *all* input databases that have up to k rows (k is set to 2 in SlabCity and can be customized by users), though cell values are not bounded. These queries will then be sent to our full verifier which we explain next.

Full verification. SlabCity incorporates state-of-the-art *full-fledged verifiers* from Cosette [26] and SPES [74] — which can prove query equivalence against *all possible inputs*. They are most suitable for common queries (such as select-project-join) and give the highest possible guarantee; as a result, they are also more expensive. SlabCity uses full verification parsimoniously, only when our tester and bounded verifier are not able to disprove a query.

4.5 Performance Ranking

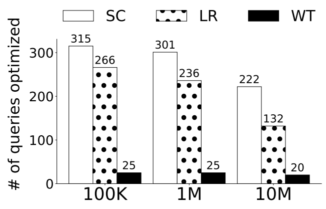
Our ultimate goal is to find equivalent queries that are *faster*. We thus use a performance ranker to select a query with the best performance among the equivalent ones. In this paper, we use `EXPLAIN` cost estimates as a proxy for the actual performance. However, SlabCity can also use the actual query latency by running candidate queries against a small sample of the data (or even the entire data, when the overhead can be well-justified, e.g., for reporting dashboards where the same query will run many times). The choice of how to estimate query performance is an orthogonal question and is not essential to our synthesis-based algorithm. In Section 5.4, we present an ablation study to see the impact of using `EXPLAIN` cost estimates on the latency of the final synthesized queries.

5 EVALUATION

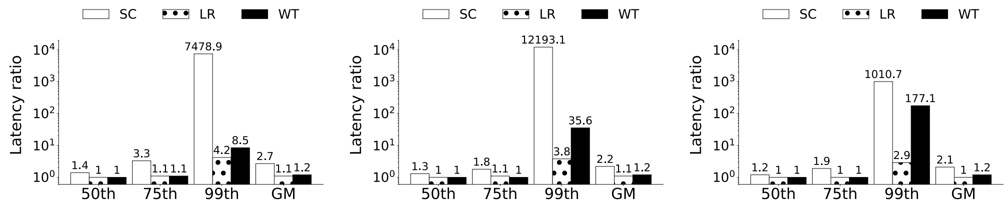
Our experiments are designed to answer the following questions:

- **Coverage:** How does SlabCity's *coverage* compare against that of state-of-the-art query rewriting techniques? That is, which technique can optimize more queries? (Section 5.2)

[8] For the complete set of rules, please see [11].

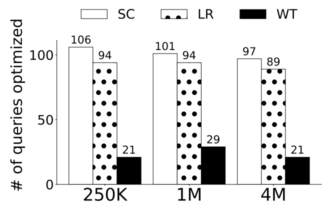


(a) Coverage (# of queries).

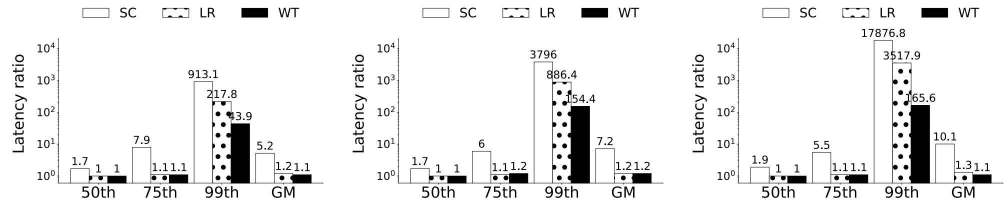


(b) Latency reduction: left is on 100K, middle is 1M, and right is 10M. "GM" means geometric mean.

Figure 6: LeetCode results across all data sizes with uniform distribution.



(a) Coverage (# of queries).



(b) Latency reduction: left is on 250K, middle is 1M, right is 4M. "GM" means geometric mean.

Figure 7: Calcite results across all data sizes with uniform distribution.

- **Query latency reduction:** How does SLABCITY compare to state-of-the-art query rewriters in terms of the efficiency of the generated queries? That is, for input queries that can be rewritten, which technique leads to faster queries? (Section 5.3)
- **Ablation study:** How important are various ideas in SLABCITY? (Section 5.4)

5.1 Experimental Setup

Testbed. All of our experiments (including running SLABCITY and baselines as well as executing queries) are conducted on Amazon EC2 r5.large instances, with 16GB RAM, 200GB gp2 SSD and Xeon E5-2686 v4 CPU running Ubuntu 22.04. We use PostgreSQL 12.13.

Workloads. We use a wide range of different workloads:

- **LeetCode.** These are SQL queries, written by actual developers to solve different problems on LeetCode [9]. Specifically, we first crawled *all publicly available* SQL queries accepted by LeetCode as "correct" solutions. However, a number of them are actually incorrect due to missing tests on LeetCode. Hence, we manually filtered out as many incorrect queries as we could. In the end, we were left with a curated suite of 1131 queries overall. Some of these solutions are poorly-written and therefore slow-running (which we hypothesize were authored by SQL novices) – this makes our LeetCode dataset especially valuable for evaluating query rewriting techniques. We also formalized the schemas and integrity constraints for all LeetCode tasks.
- **Calcite.** Our second workload is constructed from the Calcite's *optimization rules* test suite [2], which is used in prior work [68] for evaluating query rewriting. We include all 794 queries.
- **TPC.** Finally, we also use queries from the standard TPC-H [14] and TPC-DS [13] workloads.

Data generation. We generate large databases for each workload. For LeetCode, we use three data sizes (100K, 1M, and 10M). For Calcite, we use 250K, 1M, and 4M (4M is the largest data size that can fit in 16G memory). For both workloads, we follow prior work [68] and use two different data distributions: uniform and Zipfian (with a skewed parameter of 1.25, as used in [68]). We also make sure the generated data meets the integrity constraints. For TPC-H and TPC-DS workloads, we use their official data generation script with the same scale factor as in `LearnedRewrite` [76] (which is 1, meaning 1G data size).

Baselines. We compare SLABCITY against two state-of-the-art rule-based query rewriting techniques:

- **`LearnedRewrite`** (or `LR`, VLDB 2022 [8, 76]) is a state-of-the-art *rule-based* query rewriter which uses Monte Carlo Tree Search to guide the rule-based rewriting process. In particular, `LR` uses rewrite rules from Calcite and was shown to outperform multiple existing techniques from prior work [17, 49].
- **WETUNE** (or `WT`, SIGMOD 2022 [15, 68]) is the state-of-the-art automated query rewrite rule generator. It had discovered dozens of new rules, previously missing from existing rule-sets. These new rules were shown to lead to new optimizations previously not considered by existing systems (such as MS SQL Server).

5.2 Coverage

This section reports the number of queries from each workload that SLABCITY can optimize, and compare it with baselines.

Setup. Given each query Q_{in} (and the corresponding schema) from each workload, we run SLABCITY (using a 5-second timeout) and obtain an output query Q. Then, we check whether or not Q indeed optimizes Q_{in} in terms of latency: if Q has a smaller latency than Q_{in}, we say Q optimizes Q_{in}. We run each query (both input and rewritten queries) 3 times and take their average as the final latency value. Before each run, we restart the PostgreSQL service and clear the database cache. We use 10 hours as the timeout; output queries that do not terminate before timeout are marked as "not optimized". The same setup is used for the baselines.

SLABCITY can optimize more queries across different workloads.

Table 6: **LeetCode and Calcite results with Zipfian distribution.**

	Coverage (# of queries)						Latency ratio (geometric mean)					
	LeetCode			Calcite			LeetCode			Calcite		
	100K	1M	10M	250K	1M	4M	100K	1M	10M	250K	1M	4M
SlabCity	**318**	**269**	**186**	**101**	**99**	**96**	**3.1x**	**2.7x**	**2.2x**	**5.4x**	**8.8x**	**12.8x**
LR	262	243	154	95	94	86	1.1x	1.2x	1x	1.2x	1.3x	1.3x
WT	24	23	24	20	23	18	1.4x	1.6x	1.6x	1.2x	1.2x	1.2x

Results. Figure 6(a) and Figure 7(a) present the coverage results for all techinques (including SlabCity and baselines) across LeetCode and Calcite workloads (using a uniform distribution). For example, looking at Figure 6(a), for 10M, SlabCity can optimize 222 input queries, whereas `LearnedRewrite` and WeTune can only optimize 132 and 20 queries, respectively. We also note that since SlabCity's checker can fully verify only a subset of its output queries, we manually inspected all the remaining queries that only pass the tester and bounded verifier (see Section 5.5 for a detailed discussion on the limitation of full verification) and confirmed the optimizations included in our results are indeed correct (i.e., all input-output query pairs are equivalent). For the Zipfian distribution, we obtain similar results, as shown in Table 6. Overall, SlabCity significantly outperforms existing rule-based systems, highlighting the superiority of a whole-query synthesis-based approach.

For the TPC-H workload, SlabCity is able to optimize 2 (out of 22) input queries (#17 and #20) in less than 5 seconds. `LR` is able to optimize the same two queries, while `WT` cannot optimize any. The TPC-DS queries are far more complex, but SlabCity can still optimize three queries (#1, #30 and #81) within 5 seconds. `LR` can optimize two (#1, #30) but `LR`'s output queries are much slower than SlabCity's. Again, `WT` is not able to optimize any.

Discussion. The gap between SlabCity's coverage and baselines' on LeetCode is much higher than that on Calcite: we believe this highlights the usefulness of our LeetCode dataset and the advantage of our synthesis-based technique over rule-based ones in optimizing new query patterns. In particular, LeetCode has more queries that are larger and use more aggregation and window functions than those in Calcite. While SlabCity's coverage is considerably higher than all baselines, there are still some queries covered by baselines and not covered by SlabCity. Some of these queries involve operators (e.g., coalesce) that are not yet supported by our prototype. Another reason has to do with our 5s timeout. Our plan for future improvement of our prototype (e.g., supporting more operators and optimizing the performance such as by parallelizing the search) should help with both reasons.

5.3 Query Latency Reduction

In this section, we compare the performance of queries synthesized by SlabCity with those generated by (rule-based) baselines. Higher coverage (Section 5.2) means SlabCity can optimize more queries, but does it also lead to better (i.e., faster) queries than baselines?

Setup. We use the same setup as in Section 5.2 and measure the reduction ratio in query latencies across each technique's output queries. For example, given Q_{in} and its corresponding output query Q synthesized by SlabCity, a latency reduction of 5x means Q is 5x faster than Q_{in}. We still use 10 hours as the timeout — in case of a timeout, we use 10 hours as the latency; however, if both Q_{in} and Q time out, we do not include this data point.

Table 7: **Time breakdown on average and other statistics.**

% of time spent on	LeetCode	Calcite
query search (line 3)	37.4%	6.9%
checking against counterexamples (line 4)	10.8%	0.9%
equivalence checking (line 5)	51.4%	92.2%
performance ranking (line 8)	0.4%	0.1%
avg # of queries enumerated	437	104
avg # of counterexamples generated	1.4	1.2
avg # of equivalent queries synthesized	66	15

SlabCity generates faster queries across different workloads.

Results. We present the latency reduction results for LeetCode and Calcite workloads (using a uniform distribution) in Figure 6(b) and Figure 7(b). For example, looking at Figure 6(b), for 10M (rightmost figure), on average, SlabCity achieves a 50.3x latency reduction ratio *across all output queries*, whereas `LR` is 1.4x and `WT` is 12x. We also report additional statistics: median (50th) and 75th percentiles. As we can see, SlabCity always outperforms baselines by a significant margin across both workloads and for all data sizes. For output queries that can be covered (i.e., optimized) by SlabCity, the median is as high as 1.9x for LeetCode and 2.2x for Calcite. For the Zipfian distribution, SlabCity also outperforms all baselines; see Table 6 for more details.

For TPC-H, SlabCity and `LR` achieve similar latency speed-ups (both by more than an order of magnitude). For TPC-DS, SlabCity can optimize one query that `LR` is not able to optimize. For the two TPC-DS queries that both can optimize, SlabCity outputs faster queries compared to `LR`'s. For all three TPC-DS queries, SlabCity can speed-up the original queries by at least one order of magnitude.

Discussion. Similar to Section 5.2, SlabCity's margins of improvement on LeetCode are even more significant than on Calcite, which we believe is due to the same reasons mentioned earlier: existing rule-based systems and `LearnedRewrite`'s rule-set are specifically designed based on Calcite queries and LeetCode is a useful dataset to evaluate query rewriters. Careful readers may observe that SlabCity's reduction on LeetCode 10M is lower than that on 1M. This is due to 15 input queries (all from problem #1308) which do not terminate within 10 hours on both sizes. Surprisingly, SlabCity is able to optimize these queries: the fastest output query terminates in 3 seconds on 1M but takes more than 30 seconds on 10M. As we used 10 hours as the latency for the timed-out input query in both cases, this makes the reduction on 10M lower than that on 1M.

5.4 Detailed Analysis and Ablation Study

Time breakdown. Table 7 shows the breakdown of how much time on average SlabCity spends in each component (such as search and equivalence checking) and other statistics (such as the average number of queries generated during synthesis). In general, query equivalence checking (including both testing and verification) takes the majority of the time.

Synthesis efficiency. While SlabCity uses a 5s *timeout*, many of its output queries are discovered much earlier. Specifically, 58% of its

Table 8: Running time statistics to generate rewrites (in seconds).For SlabCity, we report the time when an output query (synthesized using 5-second timeout) is found.

	LeetCode		Calcite	
	median	max	median	max
SlabCity	0.84	4.84	0.74	4.81
LearnedRewrite	0.03	0.73	0.25	3.75
WeTune	0.24	0.46	0.001	0.17

output queries (generated using 5s timeout) are found within 1 second. In the words, the actual synthesis time to find the output query is shorter. Table 8 presents the median and max of SlabCity's actual synthesis time for LeetCode and Calcite queries that SlabCity can rewrite. As expected, rule-based approaches are generally faster, though LR's max time on Calcite is much slower. For TPC-H and TPC-DS, SlabCity takes 4-5 seconds for all queries it can optimize.

Impact of CEGIS. How beneficial is the use of CEGIS? That is, if we do not re-use counterexamples when disproving queries, how inefficient would SlabCity become? For LeetCode, with CEGIS, on average, SlabCity can search 437 queries in 5 seconds, where 1.4 counterexamples are generated; however, without re-using these examples, it can only search 175 queries on average within the same amount of time. This is because the tester spends extra time generating new counterexamples which are later dropped, but clearly they can be re-used to disprove other queries. This validates the observation made in prior work that CEGIS helps boost efficiency [58].

Impact of query testing. It is critical to reduce the frequency of invoking query verification. For instance, for LeetCode, on average our tester can disprove several dozens of queries within 1 second, whereas the verifier typically can handle at most a couple of queries within the same amount of time.

Impact of using dataflows to guide search. A variant of SlabCity that uses a naive search algorithm (that enumerates queries based on query size) was not able to optimize any of our input queries within 5 seconds. This highlights the importance of using dataflows to significantly accelerate query synthesis.

Impact of using cost estimates for ranking. `EXPLAIN` is known to give inaccurate cost estimates. We study a variant of SlabCity where, instead of `EXPLAIN`, we use the actual latency for each query (by running it against the database). While this variant is clearly impractical, it helps us understand how much SlabCity could be improved if we had access to a perfect predictor that could give the exact latency values. Table 9 compares SlabCity w/ exact latency against our current SlabCity w/ `EXPLAIN` in terms of coverage; Table 9 compares them in terms of latency reduction. In general, using exact latencies yields slightly higher coverage and latency reduction ratios, which is expected as that is the perfect predictor. However, we notice that using `EXPLAIN` to estimate query latency gives very close results and we believe this is a reasonable trade-off.

Percentage of fully verified rewrites. A subset of SlabCity's rewrites can be *fully* verified by the full verifier — meaning queries in this subset are guaranteed to be equivalent to their corresponding input queries, *for all possible inputs.* For example, for LeetCode 100K, 17% of the queries can be fully verified by Cosette and SPES. We explain the implications in the next Section 5.5.

Table 9: Coverage (# of queries) and geometric mean of latency reduction: using `EXPLAIN` vs. using exact latencies.

	Coverage (# of queries)						Latency ratio (geometric mean)					
	LeetCode			Calcite			LeetCode			Calcite		
	100K	1M	10M	250K	1M	4M	100K	1M	10M	250K	1M	4M
w/ `EXPLAIN`	315	301	222	106	101	97	2.7x	2.2x	2.1x	5.2x	7.2x	10.1x
w/ exact latencies	401	373	351	121	117	112	2.9x	2.6x	2.2x	4.8x	6.7x	10.4x

5.5 Discussion

While SlabCity's percentage of fully verified rewrites (about 17%) might sound low, one must remember that the vast majority of rewrite rules in existing rule-based techniques are not fully verified [25, 27] and rely on human verification, which is error-prone. In fact, a considerable fraction (about 5%) of rewrites generated by `LearnedRewrite` using the Calcite rule-set were proved incorrect by SlabCity's checker. In contrast, all of SlabCity's rewrites pass the checker. Even those few prior approaches that do offer formal guarantees, they do so for a much smaller number of queries than SlabCity (e.g., 7 queries in the FGH work [67]).

In reality, the percentage of rewrites that can benefit from full verification has little impact on SlabCity's specific use case, which, as explained in Section 1, is any scenario where a query is rerun many times, such as BI dashboard queries. In such scenarios, many rewrites receiving a "bounded verification" flag is not a hindrance since the human inspection can be performed at the time of defining the queries and before deploying the dashboard to production.

Furthermore, SlabCity's use of a tester and a bounded verifier in conjunction with full verification offers two key improvements. First, they can very effectively eliminate incorrect rewrites (by more than 80%) and hence allow human experts to focus their effort on inspecting a much smaller set of high-quality rewrites. For example, as mentioned earlier, our checker proves 5% of LR's rewrites to be wrong. Second, rewrites that pass our tester and bounded verifier are highly likely to be correct: among SlabCity's rewrites that pass, 97% of them are confirmed to be correct (either via full verification or manual inspection) and only 3% are found to be incorrect.

Finally, SlabCity is an important step in the right direction: there are many use cases that are not covered by state-of-the-art techniques, and SlabCity can discover non-trivial whole-query optimizations for many of those cases. SlabCity can also be used to augment existing approaches, whereby one can still rely on rule-based rewriting when applicable and invoke SlabCity otherwise.

6 RELATED WORK

Rule-based query rewriting. There is significant work on rule-based query rewriting [17, 24, 28, 29, 42, 50, 68]. The rules are either crafted by databases experts over decades [17], which grow very slowly and cannot handle unanticipated patterns, or discovered by automated tools [68], which can only handle a small subset of simple SQL queries. In light of the recent interest [76] in adopting deep learning in query optimization [39, 43, 70], there is also some work on using deep learning for query rewriting [76]. Given a SQL query and a set of rewrite rules, `LearnedRewrite` [76] decides the order in which the rewrite rules should be applied using the Monte Carlo Tree method with learned cost models. However, both traditional and learning techniques are fundamentally limited by the incomplete nature of rules and pattern-matching; thus, they may miss valuable rewrite opportunities (see Section 2).

Leveraging constraints for query optimization. Prior work explores semantic query rewriting by considering additional constraints. For instance, leveraging `NOT NULL` and key constraints to eliminate joins [7], using `NOT NULL` constraint to optimize queries with disjunction [5], considering key constraints to get rid of unnecessary `DISTINCT` [38], and more [4, 10]. However, they mostly rely on pattern-matching rules to identify rewriting opportunities that leverage relatively simple constraints (e.g., foreign or primary keys), while SlabCity can flexibly use more complex forms of integrity constraints during query synthesis.

Checking query equivalence. There is a body of prior work on the automatic verification of query equivalence. Cosette [26] leverages proof assistants (e.g., Coq) to interactively construct mechanized proofs for equivalent query pairs or generate counterexamples for inequivalent pairs. EQUITAS [73, 74] executes queries symbolically to generate a logical formula that encodes the query semantics, and then uses an SMT solver to verify the query. It can be applied for a subset of SQL queries (SPJ and some outer join and aggregate). SlabCity is orthogonal to these techniques as it focuses on quickly finding examples that eliminate inequivalent query candidates. On top of that, SlabCity leverages these techniques to check the equivalence of its candidate queries and provide formal guarantees of rewrite correctness.

There is also work, from the software engineering and testing literature, that uses mutation-based testing techniques to identify common failure patterns of queries [20, 57]. They use hard-coded constraint rules [57] or specifications [20] to generate test databases with the goal of killing as many query mutants as possible. These techniques consider a small query language that cannot express complex queries that arise from our workloads. EvoSQL [19] applies an evolutionary search algorithm that can test query correctness, in an offline manner, guided by the predicate coverage metric proposed by Tuya et al. [63]. In contrast, the tester is SlabCity is guided by a set of common patterns that lead to incorrect rewrites. In other words, our approach is customized to our problem domain of query rewriting and can identify counterexamples more efficiently in an online fashion during the CEIGS-based synthesis process. RATest [44] uses a provenance-based algorithm to find a minimal database that can distinguish and explain incorrect queries; however, it is quite expensive and can only support a limited SQL language, which makes it not suitable for our domain.

Program synthesis for databases. Program synthesis techniques have been used in the area of databases. For example, Blitz [40, 56] synthesizes user-defined operators from Spark programs for better parallel query execution across operators. Chestnut [69] uses synthesis to find a new data layout as well as a query plan to execute on that layout. Unlike SlabCity, they focus on non-relational queries rather than optimizing core SQL queries. Sickle [75] and Scythe [65] synthesize analytical queries for end users given input-output examples, which lowers the burden on the user to remember the finer details of SQL. Program synthesis has also been used to test database applications [47, 48, 61, 64], where the goal is to automatically construct a more diverse set of test databases in order to improve the application code coverage.

Data provenance. We also briefly discuss how data provenance is related to (and different from) the concept of query dataflow in this paper, since they might look similar to each other. There is a large body of work on data provenance in various sub-fields of database systems, such as probabilistic databases [35, 60], view maintenance [18], and explanation of query results [21, 34]. More recently, data provenance has also been used to guide the synthesis of Datalog programs [52] and SQL queries [75] from input-output examples. Data provenance is concerned with where every piece of data originates and how it is computed. In contrast, our notion of query dataflow is based on a coarser-grained, column-level flow of information within the input query. As such, computing query dataflow is significantly more tractable than data provenance, yet sufficient enough to help guide SlabCity through its search.

7 CONCLUSION AND FUTURE WORK

In this work, we presented SlabCity, the first synthesis-based query rewriting technique that is capable of *whole-query* optimization without requiring rewrite rules. SlabCity is not restricted to a given set of rewrite rules, and therefore is able to explore a larger space of candidate queries for more fruitful optimizations. In particular, our evaluation shows that, not only can SlabCity optimize many more (up to 1.7x more) queries than state-of-the-art query rewriters, but it also generates more interesting queries that are significantly faster (by up to 4 orders of magnitude).

We believe our general framework of using dataflows to synthesize rewrites is applicable to discovering rewrite rules as well akin to WeTune [68]. Specifically, given a query Q_{in} with an integrity constraint ϕ, one can first use SlabCity to obtain a faster query Q. Then, this specific transformation $Q_{in} \rightarrow_{\phi} Q$ can be generalized to a pattern $T_{in} \rightarrow_{\psi} T$ where T_{in} and T are query templates and ψ is a more general constraint such that this more general transformation still preserves equivalence. One can leverage the verifier in WeTune to check equivalence of T_{in} and T under ψ.

We also note that query rewriting is orthogonal to traditional query optimization and query tuning techniques, and thus, SlabCity can be leveraged alongside such techniques. In general, given the same amount of time and engineering resources, it should be easier to tune a rewritten query than the original one since the query optimizer starts with a more optimal query plan than it would with a poorly expressed query. In other words, the set of query plans that can be uncovered with traditional query optimization techniques is heavily limited by the starting query, which is the main reason query rewriting techniques have been a subject of much research in the field. However, a more comprehensive study to quantify the impact of better query rewriting on the effectiveness of traditional query tuning will make for an interesting future work.

ACKNOWLEDGMENTS

We thank Lin Ma, and the anonymous reviewers for the helpful feedback. This work was supported by the National Science Foundation under Grant No. CCF-2210832.

REFERENCES

[1] 49 Shocking Business Intelligence Statistics for 2021 . https://www.trustradius.com/vendor-blog/business-intelligence-statistics-and-trends (Accessed: 31 January 2023).

[2] Apache Calcite Test Suite. https://github.com/apache/calcite/blob/main/core/src/test/java/org/apache/calcite/test/RelOptRulesTest.java (Accessed: 31 January 2023).

[3] Avoiding The Most Frequent Problems in BI Projects. https://bi-survey.com/bi-avoiding-problems (Accessed: 31 January 2023).

[4] DB2 query optimization using constraint. https://github.com/prestodb/presto/issues/16413 (Accessed: 31 January 2023).

[5] Disjunctive subquery optimization. https://nenadnoveljic.com/blog/disjunctive-subquery-optimization/ (Accessed: 31 January 2023).

[6] How to Succeed in Self-Service BI. https://www.harnham.com/post/2018-10/how-to-succeed-in-self-service-bi-2 (Accessed: 31 January 2023).

[7] Join elimination. https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/ (Accessed: 31 January 2023).

[8] LearnedRewrite source code. https://github.com/zhouxh19/LearnedRewrite (Accessed: 31 January 2023).

[9] LeetCode website. https://leetcode.com/ (Accessed: 31 January 2023).

[10] PrestoDB query optimization using constraint. https://www.ibm.com/docs/en/db2/11.5?topic=performance-using-constraints-improve-query-optimization (Accessed: 31 January 2023).

[11] SlabCity technical report. https://ruidong.pl/files/slabcity-extended.pdf.

[12] The Calcite project. https://calcite.apache.org/ (Accessed: 31 January 2023).

[13] TPC-DS. https://www.tpc.org/tpcds/ (Accessed: 31 January 2023).

[14] TPC-H. https://www.tpc.org/tpch/ (Accessed: 31 January 2023).

[15] WeTune source code. https://ipads.se.sjtu.edu.cn:1312/opensource/wetune (Accessed: 31 January 2023).

[16] Meenakshi Arora and Anjana Gosain. 2011. Schema evolution for data warehouse: a survey. *International Journal of Computer Applications* 22, 6 (2011), 6–14.

[17] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data.* 221–230.

[18] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2002. On propagation of deletions and annotations through views. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* 150–158.

[19] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based test data generation for SQL queries. In *Proceedings of the 40th international conference on software engineering.* 1220–1230.

[20] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal* 24, 6 (2015), 731–755.

[21] Adriane Chapman and HV Jagadish. 2009. Why not?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 523–534.

[22] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* 34–43.

[23] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1401–1412.

[24] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A Modern Optimizer for Real-Time Analytics in a Distributed Database. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1401–1412. https://doi.org/10.14778/3007263.3007277

[25] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *arXiv preprint arXiv:1802.02229* (2018).

[26] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

[27] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.

[28] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. 2020. SQLCheck: automated detection and diagnosis of SQL anti-patterns. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2331–2345.

[29] Beatrice Finance and Georges Gardarin. 1991. A rule-based query rewriter in an extensible dbms. In *Proceedings. Seventh International Conference on Data Engineering.* IEEE Computer Society, 248–249.

[30] Goetz Graefe. 1987. *Rule-based query optimization in extensible database systems.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[31] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[32] Goetz Graefe and David J DeWitt. 1987. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data.* 160–172.

[33] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering.* IEEE, 209–218.

[34] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* 31–40.

[35] Todd J Green and Val Tannen. 2006. Models for incomplete and probabilistic information. In *Current Trends in Database Technology–EDBT 2006: EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers 10.* Springer, 278–296.

[36] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[37] Jyotsana Gupta. How slow database queries can negatively impact your business. https://wire19.com/how-slow-database-queries-can-negatively-impact-your-business/ (Accessed: 31 January 2023).

[38] Jean Habimana. 2015. Query optimization techniques-tips for writing efficient and faster SQL queries. *International Journal of Scientific & Technology Research* 4, 10 (2015), 22–26.

[39] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[40] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating Super-Operators in Big-Data Query Optimizers. *Proc. VLDB Endow.* 13, 3 (nov 2019), 348–361.

[41] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query optimization by predicate move-around. In *VLDB*. 96–107.

[42] Guy M Lohman. 1988. Grammar-like functional rules for representing query optimization alternatives. *ACM SIGMOD Record* 17, 3 (1988), 18–27.

[43] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[44] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data.* 503–520.

[45] Inderpal Singh Mumick, Sheldon J Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. Magic is relevant. *ACM SIGMOD Record* 19, 2 (1990), 247–258.

[46] M Muralikrishna et al. 1992. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB*, Vol. 92. Citeseer, 91–102.

[47] Kai Pan, Xintao Wu, and Tao Xie. 2013. Automatic Test Generation for Mutation Testing on Database Applications. In *ASE*. 111–117.

[48] Kai Pan, Xintao Wu, and Tao Xie. 2014. Guided Test Generation for Database Applications via Synthesized Database Interactions. *ACM Trans. Softw. Eng. Methodol.* 23, 2, Article 12 (apr 2014), 27 pages.

[49] Hamid Pirahesh, Joseph M Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *ACM Sigmod Record* 21, 2 (1992), 39–48.

[50] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/130283.130294

[51] Quora. What-industries-typically-have-slow-or-complex-SQL-queries. https://www.quora.com/What-industries-typically-have-slow-or-complex-SQL-queries (Accessed: 31 January 2023).

[52] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 62–1.

[53] Mark Robbins. Time is money – what is the business impact of a slow query?. https://www.linkedin.com/pulse/time-money-what-business-impact-slow-query-mark-robbins/ (Accessed: 31 January 2023).

[54] Devan Sabaratnam. Finding and fixing slow queries. https://devan.codes/blog/2021/1/17/finding-fixing-slow-queries (Accessed: 31 January 2023).

[55] Nikolay Samokhvalov. What is a slow SQL query?. https://postgres.ai/blog/20210909-what-is-a-slow-sql-query (Accessed: 31 January 2023).

[56] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP) (SOSP '17).* 631–646.

[57] Shetal Shah, S Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. 2011. Generating test data for killing SQL mutants: A constraint-based approach. In *2011 IEEE 27th International Conference on Data Engineering.* IEEE, 1175–1186.

[58] Armando Solar-Lezama. 2008. *Program synthesis by sketching.* University of California, Berkeley.

[59] Brian Stein and Alan Morrison. 2014. The enterprise data lake: Better integration and deeper analytics. *PwC Technology Forecast: Rethinking integration* 1, 1-9

(2014), 18.

[60] Dan Suciu, Dan Olteanu, Christop Koch, and Christoph Koch. 2011. *Probabilistic databases*. Morgan & Claypool Publishers.

[61] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. 2015. TesMa and CATG: automated test generation tools for models of enterprise applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 717–720.

[62] Alan Troyan. Possibly the most poorly written query in the history of mankind. https://dba.stackexchange.com/questions/96819/sql-query-possibly-the-most-poorly-written-query-in-the-history-of-mankind/96854 (Accessed: 31 January 2023).

[63] Javier Tuya, María José Suárez-Cabal, and Claudio De La Riva. 2010. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability* 20, 3 (2010), 237–288.

[64] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 425–446.

[65] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 452–466.

[66] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018. Speeding up symbolic reasoning for relational queries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[67] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Program Synthesis. (2022), 79–93.

[68] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3514221.3526125

[69] Cong Yan and Alvin Cheung. 2019. Generating Application-Specific Data Layouts for in-Memory Databases. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1513–1525.

[70] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. 931–944.

[71] James Yeh. Debugging an Unexpectedly Slow SQL Query Powering our Dashboards. https://abnormalsecurity.com/blog/debugging-slow-sql-query-powering-dashboards (Accessed: 31 January 2023).

[72] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2021. SIA: Optimizing Queries using Learned Predicates. In *Proceedings of the 2021 International Conference on Management of Data*. 2169–2181.

[73] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.

[74] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2735–2748.

[75] Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing analytical SQL queries from computation demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 168–182.

[76] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment* 15, 1 (2021), 46–58.