**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 writers 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:**
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 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 WiTune [68] and Apache Calcite [17].

**Drawback of rule-based query rewriting.** Unfortunately, the effectiveness of rule-based approaches 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 writers (e.g., WiTune [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.

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\(^5\) 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 \texttt{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 \textit{whole-query} optimization and (ii) does not require any rewrite rules. We therefore call our technique, \textsc{SlabCity}.\(^3\)

**High-level workflow of \textsc{SlabCity}**. As schematically shown in Figure 1, \textsc{SlabCity} takes a query \(Q_{in}\) and an integrity constraint \(\phi\); it returns a \textit{semantically equivalent} query \(Q^\prime\) that is likely to run faster. \textsc{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 \texttt{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 \textsc{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 \textsc{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, \textsc{SlabCity} supports an expressive subset of SQL, including nested queries and arbitrary join patterns (see the formal language definition in Section 3).

**Challenges**. However, \textsc{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 \textit{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 \texttt{HAVING SUM}), the same computations (e.g., calculating \texttt{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 pre-defined 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.

---

\(^5\)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 75-68% 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 WcTune (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. $Q_3$

Table 2: $Q_1$ is human-written. $Q'_3$ and $Q_4$ are discovered by SlabCITY ($Q_4$ is the final output as its faster than $Q'_3$).

| $Q_1$ | SELECT cid FROM (SELECT cid, SUM (revenue) OVER (PARTITION BY cid, year) AS r FROM customers WHERE year = 2021) AS tmp WHERE r > 0 |
| $Q'_3$ | SELECT cid FROM customers WHERE year = 2021 AND revenue > 0 |
| $Q_3$ | SELECT cid FROM customers GROUP BY cid HAVING SUM (revenue) > 0 |

from Table 2 is the human-written solution 5 for this task, while $Q'_3$ 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 $Q'_3$ 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. $Q'_3$ 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.

$Q_5$ 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

---

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

5While 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].
not limited to Calcite: none of the state-of-the-art rule-based query
rewriters found this opportunity,6 simply because they fail to cap-
ture 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
\[
\text{FROM } T \text{ AS } T1 \text{ JOIN } T \text{ AS } T2
\]
to find self-joins syntactically, they would still miss out on queries
expressed as
\[
\text{FROM } T \text{ AS } T1 \text{ JOIN (SELECT } * \text{ FROM } T) \text{ AS } T2
\]
In contrast, because of using query synthesis, \(Q_7\) is naturally within
\text{SLABCity}\’s search space and \text{SLABCity} can successfully find it as a
query with minimal redundancy and best performance.

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

Similar to Example 3, state-of-the-art rule-based rewrite-
tors fail to exploit the fact that both sides of \text{UNION ALL} are identi-
cal, and thus fail to eliminate the redundant computation. In fact, neither
\text{LearnedRewrite} nor \text{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 \text{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 tech-
niques were not able to optimize the input query or even rewrite
it at all, while \text{SLABCity} could (see Section 5.2). Even when they
were able to optimize the query, \text{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 \text{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
\text{SLABCity} currently supports:

- Primary and foreign keys.
- Comparisons within a row—e.g., \(T\text{.StartTime} < T\text{.EndTime}\).
- Implication constraints within a row. For example, \(T\text{.EmailType} \neq \text{spam} \rightarrow T\text{.action} = \text{NULL}\).
- 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 \{ \text{s8}, \text{iphone} \}).

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

Definition 3.1. Given a query \(Q_m\) and an integrity constraint \(\phi\),
find a query \(Q\) from our query language, such that the following
two conditions hold:

1. \(Q\) is semantically equivalent to \(Q_m\) with respect to \(\phi\). That is, \(Q\)
and \(Q_m\) produce the same output on any database \(D\) that meets
the integrity constraint \(\phi\).
2. \(Q\) runs faster than \(Q_m\). Here, query performance is measured
using \text{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 \(Q_m\) and an integrity constraint \(\phi\), and returns an output
query \(Q\) that is semantically equivalent to and faster than \(Q_m\). \(Q\) is
\(\bot\) 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 in-
ductive synthesis (CEGIS) paradigm [58] — which has found great
success in the programming languages community — for our query

---
6Similar to Calcite, LearnedRewrite [76] was only able to rewrite \(Q_8\) into \(Q_9\), but did not eliminate the self-join. WeTune was not able to rewrite \(Q_1\) at all.
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 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 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 as 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 to each query Q such that, 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, E has no counterexamples and 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 E; if not, it continues to the next candidate. If Q passes 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 Q; for the former, line 7 adds E to E. Upon termination, we rank queries in 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’ is obviously not equivalent to Q_{in}, and our equivalence checker gives a counterexample E_1 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.

<table>
<thead>
<tr>
<th>Query</th>
<th>( Q ) := an input table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \text{SELECT } l \text{ FROM } Q \text{ WHERE } \psi )</td>
</tr>
<tr>
<td></td>
<td>( \text{SELECT } l \text{ FROM } Q \text{ GROUP BY } c l s \text{ HAVING } \psi )</td>
</tr>
<tr>
<td></td>
<td>( \text{SELECT } l \text{ FROM } Q \text{ ORDER BY } c l s )</td>
</tr>
<tr>
<td></td>
<td>( \text{INNER JOIN } Q \text{ ON } \psi )</td>
</tr>
<tr>
<td></td>
<td>( \text{LEFT JOIN } Q \text{ ON } \psi )</td>
</tr>
</tbody>
</table>

Figure 2: Syntax of our query language.

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 |

<table>
<thead>
<tr>
<th>gender</th>
<th>day</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

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 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 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.
away. For example, if $Q_{in}$ involves a redundant \textsc{join}, an optimized query $Q$ may contain only dataflows from one branch of the \textsc{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 $\text{SUM}$ over a column. It is less likely that a query $Q'$ can accomplish the same goal with only \textsc{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 $\delta$ (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 $\delta$ is either an input table $T$ or a column from an input table. In the recursive cases, $\delta$ captures operations performed on top of such base dataflows. For example, $\alpha(\delta, \cdot, \cdot, \cdot)$ describes the application of an aggregate function over dataflows for argument columns. $\delta, \cdot, \cdot, \cdot$ is the composition of columns to form a table (e.g., via \textsc{select}). Expressions, such as $=$ and $\leq$, involves comparison logic, which is what $\text{op}(\delta, \cdot)$ 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 \textsc{partition by}, \textsc{order by}, \textsc{group by}, etc.

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

\[
\{ \text{scores}. \text{score}, \text{scores}. \text{day} \leq \text{scores}. \text{day}, \text{SUM}(\text{scores}. \text{score}), \cdot, \cdot, \cdot \} 
\]

We have $\text{scores}. \text{score}$ because $Q_{in}$ uses data from score column in scores table. We have $\text{SUM}(\text{scores}. \text{score})$, since $Q_{in}$ performs summation on this column. Similarly, $\text{scores}. \text{day} \leq \text{scores}. \text{day}$ 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 $\Delta$ of dataflows from a query $Q$. Figure 4 shows how to extract dataflows where we use a judgment of the form:

$Q_{ctx} \vdash \text{ALLDFS}(P) : \Delta$

This means: given a “context” $Q_{ctx}$ associated with $P$, $\Delta$ 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 $\Delta$ of a \textsc{select} query is the union of three sets: $\Delta_1$ for the query $Q$ to select from, $\Delta_2$ for the filtering condition $\psi$, and $\Delta_3$ for the target list $L$.

Conceptually, $\Delta_1$ contains dataflows manifested in \textit{all components} of $Q$: that is, if $Q$ is a nested query, $\Delta_1$ would also include dataflows from the sub-queries. Similarly, $\Delta_2$ and $\Delta_3$ capture computations performed in $\psi$ and $L$. (3) is very similar to (2) in that it also takes the union of dataflow sets for each of the arguments of \textsc{join}. The join logic is implicitly captured in $\Delta_3$ for the join condition $\psi$.

Next, let us examine the extraction rules for a condition $\psi$, which can be used as a join condition or used in \textsc{where} or \textsc{having}. Looking at (4), for a boolean combination $\text{lop}$ (e.g., $\lor$) of multiple conditions, we would first recursively invoke \textsc{allDFS} to obtain all dataflows $\Delta_1$ for each $\psi_i$: in addition, since $\text{lop}$ itself performs a logical operation, the final dataflow set also includes $\text{lop}(\delta_1, \delta_2)$: here, $\delta_i$ is the dataflow for $\psi_i$ that (different from $\Delta_i$) does not include dataflows from $\psi_i$’s components, and we use a separate function $\text{df}$ (different from \textsc{allDFS}) to compute $\delta_i$. The key difference between \textsc{allDFS} and $\text{df}$ 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 $\psi$ is an application of $\text{op}$ (e.g., $\leq$) over expressions (e.g., columns). It also makes use of $\text{df}$ to retrieve the dataflow for an expression $E_i$.

Let us dive a little deeper to see how $\text{df}$ works. (6) states that the dataflow of a logical operation (e.g., $\land$) is composed of dataflows of the two arguments $\psi_1, \psi_2$. (7) is very similar except that it concerns comparison operations (like $=$) and it invokes $\text{df}$ 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 \textsc{join} query: it recursively extracts the dataflow for $col$ given $Q$, 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 $\text{df}$ 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.

\[
s_2.\text{day} \leq s_1.\text{day} 
\]

The dataflow set of this expression is the union of three sets of dataflows: (1) the dataflows of its left argument ($\{\text{scores}. \text{day}\}$), (2) the dataflows of the right argument (namely $\{\text{scores}. \text{day}\}$), and (3) the dataflow of the comparison (i.e., $\{\text{scores}. \text{day} \leq \text{scores}. \text{day}\}$). This essentially means that the query uses data from day column of
The proof is obvious due to the monotonic nature of our data. Our key idea is to leverage the dataflow-based scoring function, which scores a program \( P \) (potentially associated with a context \( C_{\text{ctx}} \)) with respect to an input query \( Q_{in} \). First, it extracts two sets of dataflows, i.e., \( \Delta_{in} \) and \( \Delta \), for \( Q_{in} \) and \( P \) respectively using the algorithm from Figure 4. Then, it computes the set difference, i.e., \( \Delta_{\text{diff}} = \Delta \setminus \Delta_{in} \), which gives the dataflows in \( P \) but not in \( Q_{in} \). Finally, we define \( \text{score}(P|Q_{\text{ctx}}, Q_{in}) = -|\Delta_{\text{diff}}| \). 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 \( \text{score}(P|Q_{\text{ctx}}, Q_{in}) = \text{score}(Q|M) \) when \( Q_{in} \) is clear from the context.

**Dataflow-based scoring**. Now we define our score function, which scores a program \( P \) (potentially associated with a context \( C_{\text{ctx}} \)) with respect to an input query \( Q_{in} \). First, it extracts two sets of dataflows, i.e., \( \Delta_{in} \) and \( \Delta \), for \( Q_{in} \) and \( P \) respectively using the algorithm from Figure 4. Then, it computes the set difference, i.e., \( \Delta_{\text{diff}} = \Delta \setminus \Delta_{in} \), which gives the dataflows in \( P \) but not in \( Q_{in} \). Finally, we define \( \text{score}(P|Q_{\text{ctx}}, Q_{in}) = -|\Delta_{\text{diff}}| \). 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 \( \text{score}(P|Q_{\text{ctx}}, Q_{in}) = \text{score}(Q|M) \) 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 \( Q_{\text{ctx}} \), for any component \( P' \) of \( P \) (e.g., \( P' \) is a sub-query of query \( P \)), \( \text{score}(P'|Q_{\text{ctx}}) \geq \text{score}(P|Q_{\text{ctx}}) \). 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 \( P_1, \ldots, P_n \) to form a bigger program \( P \), \( P' \)'s score is never higher than the score of any \( P_i \). 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** constructing 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 \( \tilde{Q}_S \) whose score is exactly \( S \), we track only those queries \( \tilde{Q}_{S-} \) whose score is at least \( S \), because according to the monotonicity property, \( \tilde{Q}_{S-} \) is sufficient to construct \( \tilde{Q}_{S} \). 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-descending order of their scores. Line 1 initializes \( \tilde{Q}_{S=0} \) to empty set since the highest possible score is 0. Then, the loop at lines 2-8 populates 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.

**Algorithm 2 LazyEnumerate search algorithm.**

**procedure** LazyEnumerate \( (Q_{in}) \)

**input:** Input query \( Q_{in} \)

**output:** A stream of candidate queries in non-descending order of scores.

1. \( \tilde{Q}_{S=0} := \emptyset \)
2. for \( S = 0, -1, -2, \ldots \) do
3. \( \tilde{Q}_{S+1} := \tilde{Q}_{S+2} \)
4. while true do
5. \( \tilde{Q}_{S} := \text{NewQueriesWithScores}(\tilde{Q}_{S-}, \tilde{Q}_{S-}) \)
6. if \( \tilde{Q}_{S-} = \emptyset \) then break;
7. yield \( \tilde{Q}_{S} \);
8. \( \tilde{Q}_{S+1} := \tilde{Q}_{S} \cup \tilde{Q}_{S-} \)
9. end while
10. end for

**Figure 5:** Inference rules for NewQueriesWithScores.

Each \( \tilde{Q}_{S} \) for all possible scores in descending order from 0. Specifically, given \( S \), \( \tilde{Q}_{S} \) is initialized to \( \tilde{Q}_{S+1} \) (line 3). The inner loop (lines 4-8) iteratively constructs new queries \( \tilde{Q}_{S} \) with score exactly \( S \) from \( \tilde{Q}_{S+1} \) (line 5) until no more new queries are generated (line 6). Line 7 yields new queries \( \tilde{Q}_{S} \) which are then added to \( \tilde{Q}_{S+1} \) at line 8. Note that while the 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 of inference rules that describe how to construct a new query $Q'$ with score $S$ from an existing set $\overline{Q}_{\geq S}$ 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 $\psi$ to generate a JOIN query $Q'$ if $Q_1$ and $Q_2$ are both in $\overline{Q}_{\geq S}$. $\psi$ 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 $I$ 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 $\overline{Q}_{\geq S}$.

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 $\phi$, the tester returns either (1) a database $D$ which satisfies $\phi$ 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 $\phi_i$. We also encode the integrity constraint into a formula $\phi_2$. We use a constraint solver (such as Microsoft Z3) to generate satisfying assignments for $\phi_1 \land \phi_2$, where each satisfying assignment corresponds to one DB instance. If $\phi_1$ and $\phi_2$ are contradictory (i.e., $\phi_1 \land \phi_2$ 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.

**Full verification.** SLABCITY incorporates state-of-the-art full-fledged verifiers from COSSETTE [26, 66] — 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)
We use a wide range of different workloads:

- **Latency reduction**: How does SLABCity compare to state-of-the-art query rewritiers 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 rewritier 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.
Table 6: LeetCode and Calcite results with Zipfian distribution.

<table>
<thead>
<tr>
<th>Coverage (# of queries)</th>
<th>Latency ratio (geometric mean)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LeetCode</td>
</tr>
<tr>
<td></td>
<td>100K 1M 10M</td>
</tr>
<tr>
<td></td>
<td>100K 1M 10M</td>
</tr>
<tr>
<td>SLABcity</td>
<td>318 269 186</td>
</tr>
<tr>
<td>LR</td>
<td>262 243 154</td>
</tr>
<tr>
<td>WT</td>
<td>24 23 24</td>
</tr>
<tr>
<td></td>
<td>3.1x 2.7x 2.2x</td>
</tr>
<tr>
<td></td>
<td>5.4x 8.8x 12.8x</td>
</tr>
</tbody>
</table>

Table 7: Time breakdown on average and other statistics.

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

Results. Figure 6(a) and Figure 7(a) present the coverage results for all techniques (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 WE-TUNE 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 $Q_m$ and its corresponding output query $Q$ synthesized by SLABcity, a latency reduction of 5x means $Q$ is 5x faster than $Q_m$. We still use 10 hours as the timeout — in case of a timeout, we use 10 hours as the latency; however, if both $Q_m$ and $Q$ time out, we do not include this data point.

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
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 SLAB City become? For LeetCode, with CEGIS, on average, SLAB City can search 437 queries in 5 seconds, whereas the verifiers 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 verifiers 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 SLAB City 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 SLAB City 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 SLAB City could be improved if we had access to a perfect predictor that could give the exact latency values. Table 9 compares SLAB City w/ exact latency against our current SLAB City w/ EXPLAIN in terms of coverage; Table 9 compares them in terms of latency reduction. In general, using exact latencies yielding 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 SLAB City’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 COSSETTE and SPES. We explain the implications in the next Section 5.5.

5.5 Discussion

While SLAB City’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 LearnRewrite using the Calculate rule-set were proved incorrect by SLAB City’s checker. In contrast, all of SLAB City’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 SLAB City (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 SLAB City’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, SLAB City’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 SLAB City’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, SLAB City is an important step in the right direction: there are many use cases that are not covered by state-of-the-art techniques, and SLAB City can discover non-trivial whole-query optimizations for many of those cases. SLAB City can also be used to augment existing approaches, whereby one can still rely on rule-based rewriting when applicable and invoke SLAB City 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 database 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, LearnRewrite [76] decides the order in which the rewrite rules should be applied using the Monte Carlo Tree method with learned cost models. However, both

---

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

<table>
<thead>
<tr>
<th></th>
<th>LeetCode median</th>
<th>max</th>
<th>Calcite median</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLAB City</td>
<td>0.84 0.84</td>
<td>0.74</td>
<td>0.41</td>
<td></td>
</tr>
<tr>
<td>LearnedRewrite</td>
<td>0.03 0.73</td>
<td>0.25</td>
<td>3.75</td>
<td></td>
</tr>
<tr>
<td>WtTune</td>
<td>0.24 0.46</td>
<td>0.001</td>
<td>0.17</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>LeetCode</th>
<th>Calcite</th>
<th>LeetCode</th>
<th>Calcite</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100K</td>
<td>1M</td>
<td>250K</td>
<td>1M</td>
</tr>
<tr>
<td>coverage # (queries)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/ EXPLAIN</td>
<td>315 303 222</td>
<td>106 101 97</td>
<td>2.7x 2.1x</td>
<td>5.2x 7.2x 10.1x</td>
</tr>
<tr>
<td>w/ exact latencies</td>
<td>401 373 351</td>
<td>121 117 112</td>
<td>2.9x 2.6x 2.2x</td>
<td>4.8x 6.7x 10.4x</td>
</tr>
</tbody>
</table>
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. EQUIVAT [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 (SP) 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 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 WE-Tune [68]. Specifically, given a query $Q_m$ with an integrity constraint $\phi$, one can first use SLABCity to obtain a faster query $Q$. Then, this specific transformation $Q_m \rightarrow Q$ can be generalized to a pattern $T_m \rightarrow Q$ where $T_m$ and $T$ are query templates and $Q$ is a more general constraint such that this more general transformation still preserves equivalence. One can leverage the verifier in WE-Tune to check equivalence of $T_m$ and $T$ under $\phi$.

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.


