WeTune: Automatic Discovery and Verification of Query Rewrite Rules

Abstract

Query rewriting transforms a relational database query into an equivalent but more efficient one, which is crucial for the performance of database-backed applications. Such rewriting relies on pre-specified rewrite rules. In existing systems, these rewrite rules are discovered through manual insights and accumulate slowly over the years. In this paper, we present WeTune, a rule generator that automatically discovers new rewrite rules. Inspired by compiler superoptimization, WeTune enumerates all valid logical query plans up to a certain size and tries to discover equivalent plans that could potentially lead to more efficient rewrites. The core challenge is to determine which set of conditions (aka constraints) allows one to prove the equivalence between a pair of query plans. We address this challenge by enumerating combinations of "interesting" constraints that relate tables and their attributes between each pair of queries. We also propose a new SMT-based verifier to verify the equivalence of a query pair under different enumerated constraints. To evaluate the usefulness of rewrite rules discovered by WeTune, we apply them on the SQL queries collected from the 20 most popular open-source web applications on GitHub. WeTune successfully optimizes 247 queries that existing databases cannot optimize, resulting in substantial performance improvements.

Keywords: Query Rewriting, Rewrite Rule Discovery, SQL Solver.

1 Introduction

Query rewriting transforms relational database queries into equivalent but more efficient queries, which is crucial for the performance of applications supported by databases. This rewriting relies on predefined rewrite rules. In existing systems, these rewrite rules are discovered through human insight and gradually accumulated over the years. The main focus of this research is on an automatic rule generator, WeTune, which discovers new rewrite rules and identifies innovative points to optimize its existing shortcomings.

2 Related works

2.1 Traditional Query Rewrite Rules

Databases supporting web applications have become the backbone of internet applications, ranging from online shopping to banking. For many web applications, database query latency is critical to user experience.

Existing rules are often carefully designed by human experts, and may require decades of accumulation. However, relying on manual effort to discover rewrite rules is insufficient. The rich features and subtle semantics of query languages make proving equivalence and formulating rules challenging.

2.2 Research implications

WeTune, as a newly proposed query rewrite rule, offers significant improvements and advancements compared to current query rewrite rules, but it also has certain shortcomings. When reducing redundant rules, it iteratively searches, and due to the large size of the dataset, the process takes too long. Based on the research on the algorithm for reducing redundant rules, this paper proposes optimization improvements to shorten the search time and perform the reduce operation more quickly.

3 Method

3.1 Overview

The architecture of WeTune :

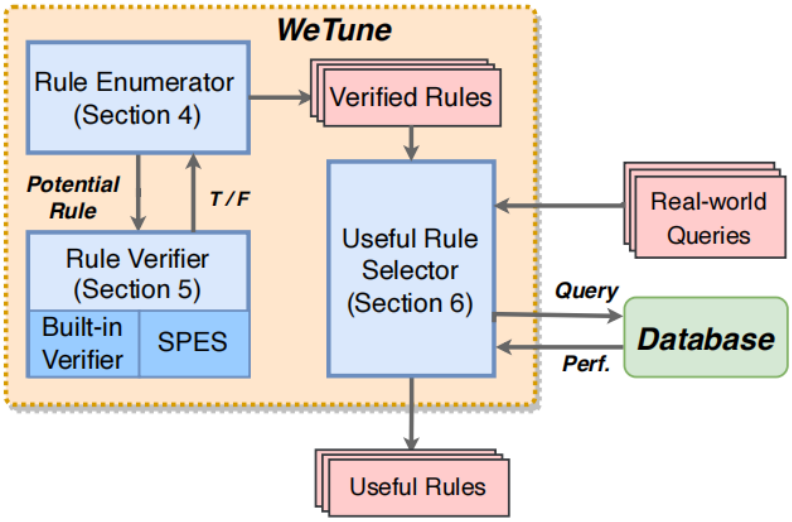

Figure 1. Overview of the method

Wetune can be broken down into three large components

Enumeration of Rules:

WTune discovers promising rules by enumerating potential rewrite rules using a rule enumerator and validating their correctness with a rule validator.

Validation of Rules:

WeTune's built-in validator first represents rules as a type of logical expression (U-expression), then converts the expressions into FOL (First-Order Logic) formulas. Finally, the FOL formulas are validated using an SMT solver.

Further Filtering of Rules:

WeTune proposes two additional optimization strategies to reduce redundant rules and eliminate `ORDER BY` clauses in SQL statements. The usefulness of committed rules is empirically determined based on the performance of rewritten queries measured in real-world scenarios.

3.2 WeTune innovates

In the process of optimizing the rule set in WeTune, the goal is to find the most permissive rule set by removing redundancy from the existing rules. The algorithm used to search for the most permissive rule set is primarily an iterative approach. However, the time required for the search is highly dependent on the size of the rule set. If the rule set is excessively large, it can lead to prolonged runtime during the program execution.

To reduce redundant rules, each rule is individually removed, and it is verified whether the same results can still be obtained. This process eliminates redundant rules to produce the most permissive rule set. However, within the entire dataset for iterative processing, most of the rules in the rule set cannot be used for rule rewriting for the concretized rules generated from the source plan template and constraints. During execution, the need to evaluate every rule against the constraints results in unnecessary runtime overhead.

To address this issue, an innovative approach is proposed: reduce the size of the dataset by grouping similar rules into a single dataset for multiple iterative runs. This reduces unnecessary runtime overhead and improves efficiency.

3.3 Reduce the drawbacks of redundant rules

During the process of reducing redundancy to obtain the most permissive rule set in WeTune (i.e., executing the reduce program), the algorithm used, as shown in the diagram, involves an iterative-like search and reduction. The time required for the search is heavily dependent on the size of the dataset used for reduction. If the rule set is excessively large, the reduction process can become time-consuming.

In WeTune, the dataset used for reduction contains over 2.3 million rows. Each row must be verified and deleted iteratively, which is the primary cause of the lengthy reduction process.

In the reduction process, rules are removed one by one, and their satisfaction of the conditional formula is verified. Through this process, redundant rules are eliminated, resulting in the most permissive rule set. However, the rule set file used for reduction contains over 2.3 million rows of rules, and each rule must be evaluated individually. During this evaluation, most of the rules in the rule set cannot be applied to rewrite the concretized query q. This results in unnecessary runtime overhead during the process, as excessive time is spent determining whether a rule RR can be reduced.

Moreover, the reduction process does not involve a single evaluation of the conditional formula. After completing a round of evaluation and deletion for the entire rule set, the resulting rule set is re-evaluated to determine whether it can be further reduced. This process is repeated until no more rules can be eliminated, at which point the reduction process is complete, and the optimized rule set is output. Consequently, the large size of the dataset significantly impacts the runtime of multiple evaluation and deletion cycles.

```
try (final ProgressBar pb = new ProgressBar("Reduce", bank.size())) {
  final List<Substitution> rules = new ArrayList<>(bank.rules());
  for (int i = 0, bound = rules.size(); i < bound; i++) {
    pb.step();

    final Substitution rule = rules.get(i);
    try {
      if (isImpliedRule(rule)) bank.remove(rule);
      else bank.add(rule);

    } catch (Throwable ex) {
      System.err.println(i + " " + rule);
      //          ex.printStackTrace();
      bank.remove(rule);
      //          throw ex;
    }
  }
}
```

Figure 2. Loose ruleset algorithms

Comparing Figures 3, 4, and 5, it can be seen that when running a rule set with 600,000 rows, only 3,194 rules can be checked for reducibility within 30 minutes. It is estimated that the first round of reduction for the 600,000-row rule set would take approximately 84 hours to complete. However, as described in Section 4.4, the method for reducing rule redundancy requires continuing the reduction process after the first round until the final output rule set size is completely equal to the input rule set size. This means that after the first reduction of the 600,000-row rule set, additional iterations are needed until the rule set becomes stable, resulting in a total runtime that far exceeds 84 hours.

In contrast, running a rule set with only 10,000 rows takes just 2 minutes and 17 seconds to complete all evaluations and produce the final rule set. During this process, the rule set was evaluated five times before reaching a stable, equivalent state. This demonstrates that the size of the rule set has a significant impact on the time required to reduce rule redundancy.

```
Reduce    0% ESC[33m??????                    ???ESC[0m   3192/531210 (0:30:25 / 83:51:30)
Reduce    0% ESC[33m??????                    ???ESC[0m   3193/531210 (0:30:26 / 83:52:40)
Reduce    0% ESC[33m??????                    ???ESC[0m   3194/531210 (0:30:32 / 84:07:37)
[Done] exited with code=null in 1908.2 seconds
```

Figure 3. 600,000 rows of ruleset reduce for example

```
Reduce  93% ESC[33m?????????????????????????????????????????????????????????????????????????????????????????????????? ???ESC[0m 8684/9323 (0:01:06 / 0:00:04)
Reduce  95% ESC[33m?????????????????????????????????????????????????????????????????????????????????????????????????? ???ESC[0m 8913/9323 (0:01:07 / 0:00:03)
Reduce  97% ESC[33m??????????????????????????????????????????????????????????????????????????????????????????????????ESC[0m 9118/9323 (0:01:08 / 0:00:01)
Reduce 100% ESC[33m??????????????????????????????????????????????????????????????????????????????????????????????????ESC[0m 9323/9323 (0:01:08 / 0:00:00)
Pass 1: 10058 -> 1887
```

Figure 4. 10,000 lines for the first reduce

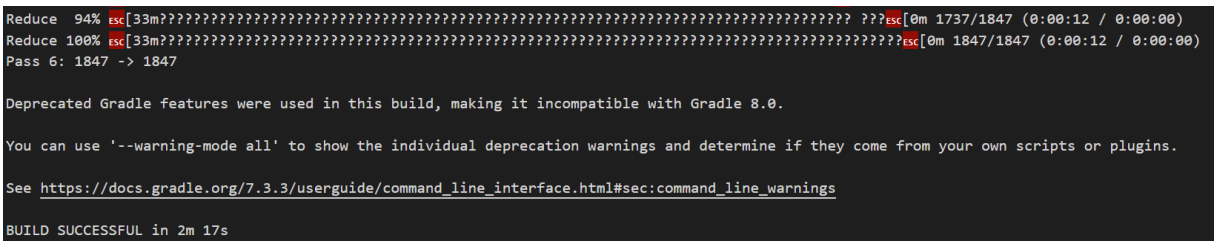

Figure 5. rows of final reduction time

To improve the shortcomings of the algorithm for reducing redundant rules, the improvements focus on addressing the root cause of these shortcomings. Specifically, since the runtime of the reduction process is closely related to the size of the rule set, reducing runtime requires reducing the size of the dataset. This can be achieved by grouping similar rules into smaller datasets and running iterative reductions on these smaller groups, thereby eliminating unnecessary computations.

Through comparisons of runtime and results obtained from reducing rule sets of various sizes, this study concludes that a rule set composed of 10,000 rules strikes a good balance between runtime and the size of the resulting rule set, making it effective for subsequent merging and further reduction. The specific steps of the proposed improvement are as follows:

1. **Sort and segment** the target rule set into 230 smaller rule sets, each containing 10,000 rules.
2. **Reduce** each segmented rule set individually, treating 10,000 rules as a unit for reduction.
3. After reducing all segmented rule sets, **merge every ten reduced rule sets** in order and perform another round of reduction.
4. **Repeat** this process until the final rule set is obtained.

With this innovation, the runtime required to reduce the entire rule set to remove redundancies is reduced to approximately 3 days. In comparison, the previous approach required about 1 month, representing a significant improvement in reduction efficiency.

```
#分割
# split -l 10000 /root/wetune/wtune_data/rules/rules.0407133816.txt -d -a 3 rules_
```

Figure 6. Split the rule set

```
for (( i = 000; i <= 230; i++)); do
if [ $i -le 9 ] ; then
    t=00$i
elif [ $i -le 99 ] ;  then
    t=0$i
else
    t=$i
fi
 out="${rules_dir}/rules$t.txt"
 in="${rules_dir}/rules_$t"
```

Figure 7. Split all rule sets

```
for ((i = 1 ; i <= 9; i++)) do
 echo rules00$i.txt;
done | xargs -i cat {} >> a_rules1.txt
```

Figure 8. Collect and merge rules in order

3.4 Verify the correctness of the innovation rules

To validate the correctness of the proposed improvement, a segment of the target rule set is extracted as an experimental rule set. The experimental rule set is then processed according to the segmentation and reduction steps described in the previous section until the final rule set is obtained. The resulting rule set is compared with the rule set obtained by directly reducing the experimental rule set to determine equivalence. If the two rule sets are equivalent, it confirms the correctness of the proposed improvement.

The main code for performing the equivalence check between the two final rule sets is shown in Figure 9.

```
for Text1 in `cat  uniq1`
do
        for Text2 in `cat uniq2`
        do
                if [ "$Text1" =  "$Text2" ]
                then
                echo $Text1 >> thesame
                fi
        done
done
```

Figure 9. The rule set is sorted and judged on a line-by-line basis

In Figure 10, the experimental rule set contains 10,000 rows. It is first reduced as a whole, and the resulting rule set is sorted and stored in `uniq1`. Then, the experimental rule set is split into two segments of 5,000 rows each, which are individually reduced. The resulting two reduced rule sets are then merged and reduced again, with the final rule set sorted and stored in `uniq2`. After computing the `md5sum` of both files, the encoded values are found to be identical. Additionally, the number of rules in the `thesame` file matches the number of rules in both `uniq1` and `uniq2`, proving that the two rule sets are equivalent.

To ensure that this result is not coincidental, further experiments were conducted with experimental rule sets containing 5,000 rows, 20,000 rows, and 30,000 rows. Each rule set underwent the same segmentation, merging, and reduction process, followed by equivalence checks. The results were consistent with those in Figure 2, with all cases proving equivalence. Each experiment was repeated more than three times to eliminate the possibility of coincidence, thereby validating the correctness of the proposed improvement.

```
root@5714bd077564:~/wetune/wtune_data/rules# md5sum uniq1 uniq2
ed6abdd66d18aad4e4a6251abf8b1891  uniq1
ed6abdd66d18aad4e4a6251abf8b1891  uniq2
```

Figure 10. The rule set is directly judged after sorting

4 Implementation details

4.1 Example of a template enumeration

WeTune can enumerate up to 4 operators at most, excluding the Input operator. The underlying principle is primarily explained in Section 4.2, which introduces the enumeration principle for templates.

As shown in the figure, it illustrates an example of WeTune enumerating templates with up to 4 operators in the tree structure. The enumerated templates are subsequently paired to form `<qsrc, qdest>` pairs,

which provide conditions for the subsequent constraint enumeration.

All templates with a maximum size of 4 operators are printed, resulting in a total of 3,113 templates.

```
InnerJoin(Proj*(InnerJoin(LeftJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(LeftJoin(InnerJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(LeftJoin(LeftJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(InSubFilter(InnerJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(InSubFilter(LeftJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(InSubFilter(InSubFilter(Input,Input),Input)),Input)
LeftJoin(Input,Proj(InnerJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InnerJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj(LeftJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj(LeftJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InSubFilter(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InSubFilter(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InSubFilter(InSubFilter(Input,Input),Input)))
LeftJoin(Input,Proj*(InnerJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InnerJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(LeftJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(LeftJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InSubFilter(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InSubFilter(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InSubFilter(InSubFilter(Input,Input),Input)))
```

Figure 11. part of running result

4.2 Query rewrite example

The rules generated by WeTune are applied to actual query statements. For each rule, the following items will be printed:

1. **The rule itself**, as shown in item 1 in the figure.
2. **A query q0**, which the rule can be applied to.
3. **The rewritten result q1**, obtained by applying the rule to q0, as shown in item 2 in the figure.

For rules that can be proven by WeTune's built-in validator, additional items will be printed:

1. **A pair of U-Expressions**, translated from the rule, as shown in item 3 in the figure.
2. **One or more Z3 fragments**, formulas submitted to Z3 for UNSAT verification, as shown in item 4 in the figure.

The figure includes a special constraint relationship, **NotNull(r, a)**, which is part of WeTune's handling of NULL values. WeTune introduces a new predicate, **IsNull**, for U-Expressions. When x is NULL, **IsNull(x)** returns true, with **[IsNull(x)] = 1**. In FOL relationships, **[IsNull(x)]** is represented as **NotNull(r, a)**, as shown in Table 4.3.

The figure demonstrates multiple Z3 fragments used for proof, where formulas are submitted to Z3 multiple times to verify that each fragment is UNSAT. This approach is intended to reduce the burden on Z3. When

proving that **(p0 p1 ... pn) (q r)** is UNSAT, it is broken down into smaller proofs, such as proving **(p0 p1 ... pn) q** and **(p0 p1 ... pn) r** are both UNSAT.

This is particularly relevant when applying theorem **Formula 4.4**. Detailed explanations of this principle are provided in Section 4.3. Rules involving related constraints require all Z3 fragments to be UNSAT to confirm the correctness of the rule.

As for why UNSAT is proven, Section 4.3 explains this as well: to prove the equivalence of an FOL formula, demonstrating that an FOL formula is unsatisfiable (UNSAT) is much easier. This is because SMT solvers halt as soon as they detect an UNSAT contradiction, avoiding exhaustive reasoning.

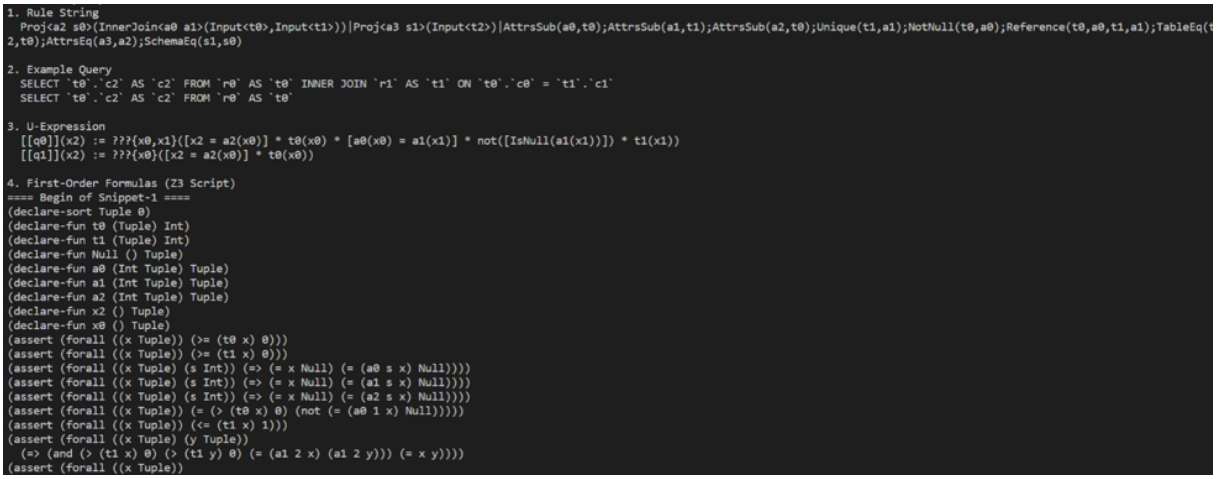

Figure 12. Query rewrite example

```
    (and (= (a0 1 x0) (a1 2 x1)) (not (= (a1 2 x1) Null)) (> (t1 x1) 0)))))
(assert (not (= x1 x1_)))
(assert (= (ite (= x2 (a2 1 x0)) (t0 x0) 0) (ite (= x2 (a2 1 x0)) (t0 x0) 0)))
(assert (and (= x2 (a2 1 x0)) (> (t0 x0) 0)))
(assert (Z x1))
(assert (Z x1_))

(check-sat)
==== End of Snippet-5 ====
==> Result: UNSATISFIABLE

Rule-7: EQ
```

Figure 13. Query rewrite example

4.3 Constraint enumeration example

The principle of constraint enumeration is mainly explained in Section 4.2. The example will list the constraints between the plan templates of the given rules and search for the loosest set of constraints. Each constraint set being checked, along with its verification results, will be printed. The found rules and metrics will be added after the enumeration is completed. The final expected rules that meet the conditions will be printed out.

The following diagram illustrates the process of constraint enumeration. The constraints that can be proven as equal (EQ) will be placed into the EQ cache, while those that are unknown will be considered as unable to

prove equality. Finally, a summary of the enumeration results will be output, including the following:

1. The size of the C* constraint set.
2. The number of enumerated constraint sets.
3. The number of EQ constraint sets.
4. The number of NEQ constraint sets.
5. The number of unknown constraint sets.
6. The number of calls to the built-in verifier.
7. The number of EQ cache hits.
8. The number of NEQ cache hits.
9. The number of relaxed EQs.

The sum of the EQ constraint sets, NEQ constraint sets, unknown constraint sets, EQ cache hits, and NEQ cache hits equals the total number of enumerated constraint sets.

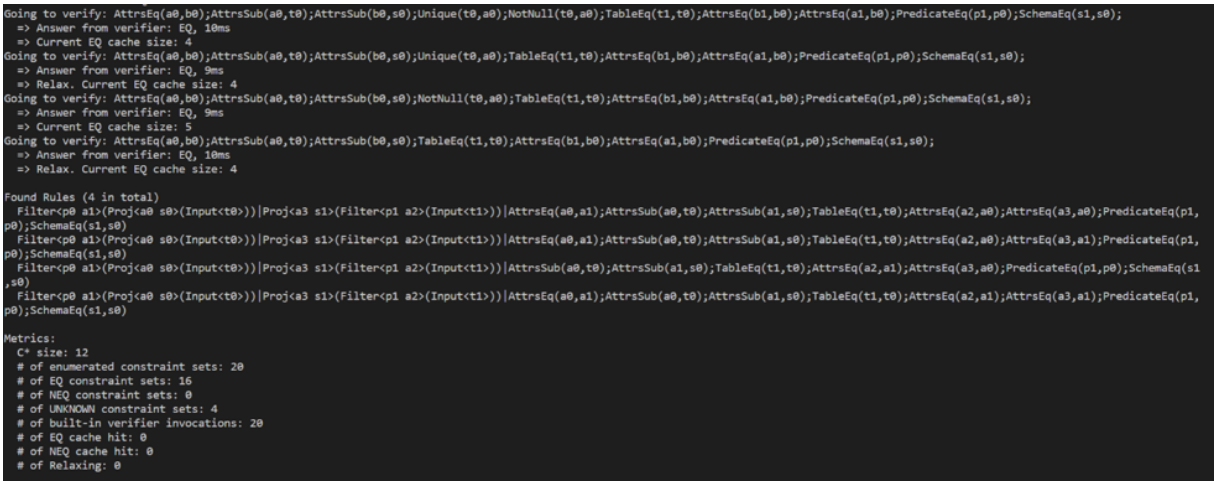

Figure 14. Constraint enumeration example

5 Conclusion and future work

This paper presents WeTune, which can automatically discover the rewrite rules for SQL queries. It enumerates all valid logical query plans up to a certain size to discover equivalent plans based on a new SMT-based verifier. We apply the rules discovered by WeTune on SQL queries collected from the 20 most popular opensource web applications on GitHub. WeTune successfully optimizes 247 queries that existing databases cannot optimize, resulting in substantial performance improvements

References

[1] Douglas Barbosa Alexandre. 2018. Improve the query performance to find unverified projects. https://gitlab.com/gitlab-org/gitlab/-/commit/11e93a9a4c2ac1b5 bd4d32a93a949fc8afbcc449?merge_request_iid=5348

[2] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 394–403.

[3] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https: //doi.org/10.1145/3183713.319066

[4] Andreas Brandl. 2018. Replace OR clause with UNION. https://gitlab.com/gitlaborg/gitlab-foss/-/merge_requests/17088#note_59749778

[5] Apache Calcite. 2021. Calcite Test Suite. https://github.com/georgia-tech-db/ spes/blob/main/testData/calcite_tests.json.

[6] 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.

[7] Hugh Darwen Chris J Date. 1996. *A Guide to the SQL Standard, Forth Edition*. Addison-Wesley Professional. https://www.amazon.com/Guide-SQL-Standard- 4th/dp/0201964260

[8] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (July 2018), 1482–1495. https://doi.org/ 10.14778/3236187.3236200

[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research* (Chaminade, California, USA) *(CIDR '17)*.

[10] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. *SIGPLAN Not.* 52, 6 (June 2017), 510–524. https://doi.org/10.1145/3140587