A Framework for Testing Query Transformation Rules

Hicham G. Elmongui*
Purdue University
elmongui@cs.purdue.edu

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

In order to enable extensibility, modern query optimizers typically leverage a transformation rule based framework. Testing individual rule correctness as well as correctness of rule interactions is crucial in verifying the functionality of a query optimizer. While there has been a lot of work on how to architect optimizers for extensibility using a rule based framework, there has been relatively little work on how to test such optimizers. In this paper we present a framework for testing query transformation rules which enables: (a) efficient generation of queries that exercise a particular transformation rule or a set of rules and (b) efficient execution of corresponding test suites for correctness testing.

Categories and Subject Descriptors

H.2.4 [**Database Systems**]: Query Processing

General Terms

Algorithms, Measurement, Performance.

Keywords

Database Testing, Query Optimization, Transformation rules

1. INTRODUCTION

Query optimizers in today's DBMSs are responsible for obtaining a good execution plan for a given query. Since a query optimizer plays a crucial role in determining the performance of a query, it is very important to rigorously test the optimizer to ensure that it functions correctly. There has been extensive work on how to architect query optimizers in order to make them extensible (e.g., [12][13][16]) using a rule based framework. However, there has been relatively little work on how to effectively test such query optimizers. It is well recognized that testing is an integral part of any development cycle and typically more than 50% of the entire development cycle is spent in testing [2].

Testing the query optimizer has several dimensions which include accuracy of cardinality estimation and costing modules, the search space of the optimizer etc. In this paper we focus on query optimizers that use a rule-based architecture. Examples include industrial query optimizers such as IBM Starbust [16], Microsoft SQL Server [13], Tandem's NonStopSQL [7] as well as academic prototypes such as the Volcano optimizer [12]. Such optimizers use transformation rules as the basic primitive in order to generate different alternative plans for a query. The set of transformation rules (e.g., join commutativity and associativity, pushing Group-By below join etc.) used by an optimizer largely determines the search space of plans considered by the optimizer and thus is a key factor in determining the quality of the final plan. While problems related to testing the components of the optimizer such as the cardinality estimation and costing modules remain pertinent for a rule based optimizer, in this paper we focus on issues related to testing the transformation rules.

One way to broadly categorize the issues that arise in the context of rule testing is as follows: 1) *Coverage*: Ensure that a transformation rule has been exercised during query optimization in several different queries. 2) *Correctness*: Ensure that when a transformation rule is exercised for a query, it does not alter the results returned when the query is executed. 3) *Performance*: Analyze how the transformation rule impacts the performance of a query/workload. In this paper, we focus on the first two aspects, namely coverage and correctness.

From the perspective of *rule coverage*, it is desirable to have tests cases in the form of SQL queries such that when the queries are optimized, they exercise all rules. In addition to ensuring that each rule is exercised, it can also be important to test that pairs of rules (in general, a set of rules) are exercised together in a query – to help capture rule interactions. Although the rule coverage problem is important, there is little previous work in this area. The state-of-the-art approach is to use stochastic methods to generate SQL queries (e.g. [1][17]) until we find a query that exercises the desired rule or rule pair. Such a trial-and-error approach has the problem that it can take many trials to even find a single query that exercises the given rule or rule pair, and rule coverage testing requires finding several such queries. This is compounded by the fact that such randomly generated queries tend to be rather complex, and thus optimizing the query in each trial can take a large amount of time. Another alternative is to build APIs that support manual generation of SQL queries [9]; however this approach can be too time-consuming and simply does not scale to crucial scenarios such as pair wise (or larger) rule interactions.

We note that in general the above problem of generating a query such that a given rule (or set of rules) is exercised, is very challenging. This is because it is hard to precisely capture the *sufficient* conditions for a rule to be exercised by the optimizer. For example, modern optimizers use pruning steps in the optimizer's search algorithm that discards a rule based on constraints or heuristics. As one illustrative example, consider the rule that pushes down a Group-By Aggregate over a join [3]. This rule is exercised only if certain functional dependencies are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29-July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06…$5.00.

* Work done while visiting Microsoft Research. The author is also affiliated with Alexandria University, Egypt.

guaranteed to hold in the join result (e.g., the Group-By must include the joining columns).

An important contribution of this paper is a technique for query generation that can address some of the above limitations. Our technique leverages the intuition that for a particular rule to be exercised during query optimization, the input logical query tree must contain a "pattern" corresponding to that rule ([12][16]). In other words, existence of such a pattern in the logical query tree is a *necessary* (but not a sufficient) condition for the rule to the exercised. For instance, for the rule that pulls up a Group-By operator above a join operator, we know that a necessary condition for the rule to be exercised is that the input logical tree should contain a join operator with a Group-By operator in one of its subtrees. We leverage the above intuition by extending the DBMS with a new API that exports such patterns. This in turn, enables the query generation method to directly leverage these rule patterns while generating the SQL query. Although the above technique does not guarantee that the generated query will exercise the rule, our experiments indicate that our technique can dramatically improve the number of trials (and hence the time) required to create test cases for rule coverage.

Another important scenario is *correctness testing* of rules. One approach for testing the correctness of the rules uses the following methodology. For each randomly generated query, check which rules were exercised during query optimization. For each such rule: (a) execute the original query and obtain the results and (b) execute the plan obtained for the original query with the corresponding rule turned off (i.e. disabled), and then check if the results of the query are identical or not. Naturally, if the results are not identical, it indicates a correctness bug. In order to have sufficient confidence in the correctness of a rule, we may need to perform this validation over several (say **k**) such randomly generated queries for each rule. Since this methodology requires *executing* queries for correctness validation, the time taken to run these test suites can be significant. Thus the key challenge in such correctness validation is *efficiency*, i.e. improving the time taken to execute the test suites.

A second contribution of this paper is that we show how to significantly reduce the time taken to execute a test suite for correctness testing of transformation rules. We exploit the following key observations: (a) When a query is optimized, often *multiple* rules are exercised. (b) The cost of a query when a rule is turned off can sometimes be much higher than the cost of the query when the rule is turned on. We introduce the novel problem of *test suite compression*: Given an initial set of randomly generated queries, we identify the best way to map queries to rules such that the time taken to run the entire test-suite is minimized (while maintaining certain invariants). In this paper, we study one version of this problem (where the constraint that the number of distinct queries (**k**) validated for each rule remains the same, we discuss another version of the problem in Section 7). We show that the test suite compression problem is NP-Hard, and present algorithms for it including a constant factor approximation of the optimal solution. Our experimental evaluation confirms our intuition that the above optimization can indeed help significantly reduce the time required for correctness testing of rules.

In Section 2, we present a brief overview of rule based query optimizers and introduce our framework for transformation rule testing. In Section 3, we discuss our approach to the query generation problem. In Section 4, we introduce the test suite compression problem; and present algorithms for it in Section 5. We present experimental results in Section 6, discuss extensions in Section 7, related work in Section 8, and conclude in Section 9.

2. PRELIMINARIES

2.1 Transformation based Query Optimizers

In this paper we consider query optimizers that use a transformation rule based architecture as described in [12] [13]. Such a framework has been used to build both industrial query optimizers (e.g. Tandem's NonStop SQL [7] and Microsoft SQL Server) as well as optimizers used in academic prototypes (e.g. the Volcano optimizer [12] and the Columbia optimizer [4]). In this section we give a brief overview of a ruled-based framework for query optimization (see [13] for more details).

Transformation rule-based optimizers use a top-down approach to query optimization. The optimizer is initialized with a logical tree of relational operators corresponding to the input query. The goal of the optimizer is to transform the input logical tree to an efficient physical operator tree that can be used to implement the query. For this purpose, *transformation rules* are used to generate different alternative plans for executing a query. The set of rules that are available to the optimizer essentially determines the search space of plans considered by the optimizer and thus is a key factor in determining the quality of the final plan.

There are two main kinds of transformation rules. *Exploration rules or logical transformation rules*, when exercised, transform logical operator trees into equivalent logical operator trees. Some examples of exploration rules include join commutativity and pushing group by below join. *Implementation rules or physical transformation rules*, when exercised, transform logical operator trees into hybrid logical/physical trees. Example implementation rules include rules that transform a logical join into a physical hash join.

We note that other extensible optimizers such as Starburst [16] also leverage the idea of transformation rules during the query rewrite phase to generate alternative logical representations of the input query. In principle, the techniques described in this paper can be extended to such optimizers even though they are not based on the Cascades framework.

2.2 Definitions

We use the following definitions and notations in the paper:

Set of transformation rules: We denote the set of transformation rules for the optimizer by $\mathcal{R} = \{r_1, \ldots r_n\}$.

RuleSet for a query: When a query q is optimized, we denote the subset of transformation rules that are exercised as RuleSet(q).

Execution plan and cost: For a given query q, we use *Plan*(q), and *Cost*(q) to refer to the execution plan chosen by the optimizer and its cost respectively. Let $R \subseteq \mathcal{R}$ be a set of rules. We denote the execution plan and cost of a query q when the set of rules R is disabled (i.e. turned off) by *Plan*(q, ¬R) and *Cost*(q, ¬R).

Logical query tree: A logical query tree is a tree of logical relational operators where each operator has been instantiated with its arguments.

Figure 1 shows an example of a logical query tree, where the two leaf operators Get(T1), Get(T2) refer to accessing relations T1 and T2 respectively. Similarly, the join and projection operators also contain the respective arguments.

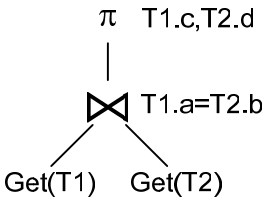

Figure 1. **Example of a logical query tree.**

2.3 A Framework for Testing Rules

Since transformation rules are a critical component of the query optimizer, testing individual transformation rules and their interaction is an important part of testing the overall query optimizer. In this paper we assume that we are given as input a test database, i.e., the database is fixed. The techniques we present are therefore general in the sense that they can be invoked against any database.

There are at least two key aspects to rule testing. One important aspect is from the perspective of *rule coverage* i.e. we would like to have test cases in the form of SQL queries where a given rule (or a given set of rules) is exercised. This is important for code coverage which can ensure that the code corresponding to the rules have been covered. Observe that this does not require execution of the query. It relies on optimizing the query and requires the ability to track which rules are exercised during query optimization.

A second aspect is *correctness testing* of the rule. While testing cannot in general, prove that a transformation rule has been correctly implemented in the DBMS, it is possible to find test cases where the rule has *not* been correctly implemented. One methodology for finding such correctness bugs for a rule is to check that the results produced by a query when the rule is exercised are identical to the results of the same query when the rule is *not* exercised. This requires: (a) The ability to turn on/off a given rule during query optimization; (b) Executing the two plans (when they are different). By repeating this methodology for several different randomly generated queries (e.g. generated via a stochastic method), we can increase confidence in the correctness of the rule. Unlike the case of code coverage, the queries used for validating correctness need to be executed. Thus, efficiency of executing the above queries for correctness testing is a key challenge.

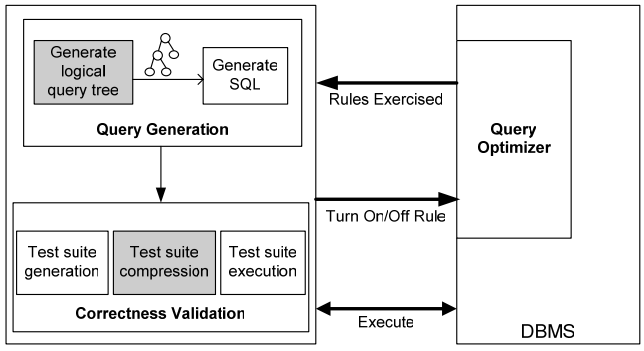

Figure 2. Overiew of architecture

In this paper, we describe an initial framework for testing transformation rules that can address the above scenarios. The key components of our framework are shown in Figure 2. We now provide an overview of each of the components.

Query Optimizer Extensions: We assume a query optimizer with support for the following functionality. First, is the ability to track which rules are exercised during query optimization. Using this extension allows us to determine RuleSet(Q) for any query Q.

Second, we support the ability to optimize (and execute) a query when a given set of transformation rules is turned off. In other words, this extension enables obtaining *Plan*(Q, ¬R) for any Q and set of rules R. We note that many existing optimizers may already have support for one or both of these extensions.

Query Generation: The query generation component takes as input a set of rules R, and generates a SQL query such that *all* rules in R are exercised when that query is optimized. Such a module is useful for both code coverage as well as correctness validation. We identify two key modules for query generation. The first module, which we refer to as *Generate Logical Query Tree* generates a logical query tree that can potentially exercise a given rule or rule pair. This is shown as the shaded box in Figure 2, and is the focus of Section 3. The second module, which we refer to as *Generate SQL*, takes as input a logical query tree (see Figure 1 for an example) and generates a SQL statement corresponding to the query tree. We use a module whose functionality is similar to one presented in [9], and therefore we do not focus on it in this paper.

Generate Logical Query Tree: Observe that the problem of generating a logical query tree such that a given rule or rule pair is exercised is non-trivial. Manually generating a logical tree that is guaranteed to exercise a rule or rule pair is both difficult and time-consuming, and does not scale with the number of rules. For example, if there are 25 transformation rules, generating test cases for all ${}^{25}C_2$ rule pairs manually is not feasible. The alternative trial-and-error approach of using randomly generated queries (e.g. as in [1][17]) is also not adequate since: (a) It is inefficient, i.e., it can require many trials before a randomly generated query exercises a given rule (or rule pair). (b) Randomly generated queries can be hard to interpret. For debugging and understandability purposes, it is desirable to generate a query with a small number of logical operators such that the rules in R are exercised. Thus a key challenge is to *efficiently* generate a logical query tree with a *small number of operators* that exercises a given rule. The efficiency of this module can be measured by the number of trials (and/or time) required to find a query that exercises the rule. Logical query tree generation for exercising rules is the subject of Section 3.

Finally, note that logical query tree generation module can also be extended for generating more complex queries that exercise a given rule. To enable such scenarios, the above module exposes the ability to add an additional the number of (random) operators to an existing logical query tree as a constraint (e.g., generate a logical query tree with 10 operators that exercises a given rule). For instance, such queries are useful for correctness testing.

Correctness Testing: Correctness testing can be performed for singleton rules, rule pairs or in general over any subsets of rules. In this paper, we focus on singleton rule and rule pairs since these are the most fundamental cases that need to be covered. We present the discussion below for singleton rules, but the arguments carry over to rule pairs as well. To validate correctness of each rule, we need to generate **k** distinct queries, each of which

exercises (at least) that rule. We refer to these queries as the *test suite* for a singleton rule $\{r_i\}$, denoted by TS_i. If there **n** singleton rules, we require **k** distinct queries for each of the **n** rules. Thus, the overall test suite for all rules is: $TS = \bigcup_{i=1..n} TS_i$

The *Test Suite Generation* module generates a test suite as described above for a given set of rules (**k** is a parameter to this module). Queries in the test suite can be generated by invoking the Query Generation module described previously.

For a given test suite, the *Test Suite Execution* module executes the test suite as follows. For each query q in TS_i, we execute Plan(q) and Plan (q, $\neg\{r_i\}$) (the plan obtained when we disable rule r_i) and check if the results of executing the two plans are identical. Thus, the total cost of executing a test suite is[1]:

$$Total\,Cost = \sum_{i=1}^{n} \sum_{q \in TSi} Cost(q) + Cost(q, \neg\{r_i\})$$

We refer to the above technique of generation and execution of a test suite, where **k** distinct queries are generated and executed for each rule independently, as the BASELINE method. Since the queries have to be executed, the time taken for the BASELINE method can be significant. Thus a key question is whether the efficiency of correctness testing can be improved significantly while still ensuring that each rule is validated for **k** distinct queries. The *Test Suite Compression* module (in Figure 2) addresses this problem. In particular, the test suite compression step identifies a subset TS' $\subseteq$ TS, while satisfying the constraint that TS' contains for each rule, **k** distinct queries where that rule is exercised. The objective is to minimize the cost of executing the test suite thereby substantially improving upon the BASELINE method. Test suite compression is the focus of Sections 4 and 5.

We begin by first discussing the query generation problem in Section 3; in particular the logical query tree generation problem.

3. LOGICAL QUERY TREE GENERATION

As mentioned earlier, the problem of testing that a rule has been exercised can be viewed as a query generation problem: Given a transformation rule, we need to generate a SQL query which exercises the rule when optimized. In this section, we first highlight some of the challenges, and then present our approach to the query generation problem.

The key challenge in generating a query that exercises a particular rule is that it is difficult to precisely capture the *sufficient* conditions for the rule to be exercised. In general, the exact preconditions necessary for a rule to be exercised can be arbitrarily complex. For example, the search algorithm used by the optimizer could discard a rule based on constraints/heuristics. There could also be cases of rule dependencies, where the exercising of one rule occurs only when one or more *other* rules are first exercised. For example, consider the input logical query tree: R Join (S LOJ T), where LOJ stands for left outer-join. Consider the following two rules: (1) Associativity of Join and Outer-join. (2) Join commutativity. We know that in general outer-joins and joins do not commute. However, if the join predicate is between R and S, then the first rule can be exercised, which results in a logical tree (R Join S) LOJ T. Observe that the second rule can now be applied on (R Join S).

The state-of-the-art approach for query generation (e.g., [1][17]) is to keep generating queries using a stochastic process until one finds a query that exercises the required transformation rule. Note that we can track which optimizer rules actually were exercised during query optimization by using the RuleSet interface (Section 2.2). We note that none of the previous work has however focused on generating queries that exercise a certain transformation rule in the query optimizer. As discussed earlier, the above trial-and-error approach can require many trials before it finds a query that exercises the given rule. For instance, consider a transformation rule that pulls up a Group-By operator over a left outer-join. Obviously, a randomly generated query is not likely to succeed unless it happens by chance to include a Group-By *and* a left outer-join in the same query. Thus, the random generation approach can require a large number of trials before it finds an appropriate query.

In this section, we study how we can significantly improve upon the state-of-the-art for this problem. Our key observation is that we can leverage rule patterns that serve as a *necessary* (although not *sufficient*) condition for a transformation rule to be exercised in the query optimizer for the purpose of query generation. In most cases, this significantly reduces the number of trials needed to find a query that exercises the given rule(s). We present the discussion below for the case of a singleton rule. We discuss extensions to support rule pairs in Section 3.2.

3.1 Exploiting Rule Patterns

Rules in a transformation based optimizer can be in general be represented by the triple (*Rule Name, Rule Pattern, Substitution*) [13]. During query optimization, the rule engine checks if the input logical tree matches the Rule Pattern. If so, it invokes the Substitution function that generates a new logical tree that should be included as part of its search. Thus a necessary condition for a rule to be exercised is that the logical tree considered during the search contains the pattern of the corresponding rule.

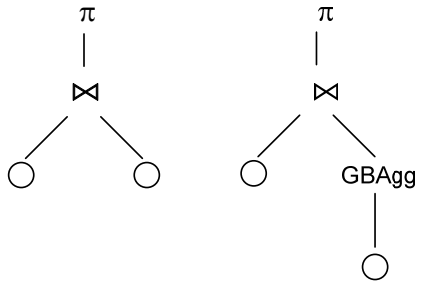

Figure 3. Example Rule Patterns

Figure 3 illustrate examples of rule patterns for two transformation rules: the join commutativity rule, and a rule for pulling a Group-By Aggregate above a join operator.

As the figure indicates, the rule patterns include operators that must be present (such as the Join and the GBAgg operator in the second example) as well as placeholders for *generic* operators (represented by circles in the patterns). These generic operators can match any logical operator. Thus, for the first rule pattern (for the join commutativity rule) to be exercised, the input logical

[1] Note that if Plan(q) and Plan(q, $\neg\{r_i\}$) are identical, it is not necessary to execute the query since the results are guaranteed to be the same.

query tree that should have a join operator (irrespective of what its children are).

Recollect that the Generate Logical Query Tree module (Section 2.3) takes as input a rule and outputs a logical query tree. Below, we described how this is achieved. We have extended the database server with an API through which it returns the rule pattern tree for a rule in a XML format. To generate a query that exercises a particular transformation rule, the query generation module first builds a logical query tree starting with the rule pattern and: (a) instantiates actual operators in place of the generic operators. For example, for the join commutativity rule, we can instantiate each of the generic operators with *Get* operators. These are leaf operators that correspond to accessing base relations. (b) Once the operators are instantiated, we select the *arguments* for each operator. For example, the Get operators can be instantiated with relations T1 and T2 respectively as their arguments. Similarly, the join operator can be instantiated with a join predicate such as T1.a = T2.b as its argument. Thus, at the end of this step we have generated a valid logical query tree (e.g. a tree such the one shown in Figure 1). Finally, the Generate SQL module (Section 2.3) is invoked with the above logical query tree to generate a valid SQL statement. Note that rule patterns can also provide sufficient conditions for implementation rules to be exercised. For example, for the hash join implementation rule to be exercised, the input pattern would need to include a join operator node. Thus, the idea of leveraging rule patterns from the optimizer can enable us to automatically generate queries that exercise a particular rule.

As mentioned above, in general, a logical query tree that contains a rule pattern is not sufficient to guarantee that the particular rule is exercised during optimization. For example, for the rule that pulls up the Group-By Aggregate over a join, some additional conditions are required to hold (for e.g., the join predicate does not reference the aggregate results). However, observe that if such constraints are well abstracted in the database engine, they can potentially be added as additional preconditions on the input pattern and leveraged by the query generation module. Certain rules may also require that certain constraints on the schema or the data instance hold in order to guarantee that it is exercised; we discuss such cases in Section 7.

Despite the fact that leveraging a rule pattern does not guarantee that a rule is exercised, for the set of transformation rules used in our experiments (Section 6), we observed that by exploiting the basic rule patterns in query generation, we can significantly reduce the number of trials required compared to the random query generation method.

Finally, it is interesting to note that if despite the use of the rule pattern we are not able to find a query that exercises that rule, it could be an indication that the rule is dependent on other rules being exercised. We plan to study such handling such rule dependencies as part of future work.

3.2 Extensions for Rule Pairs

So far, we have focused on the query generation problem for singleton rules. In addition to testing single rules, it is also important to test pairs (in general, a set) of rules to cover rule interactions. In Section 3.1, we outlined how to leverage the rule patterns that are used during optimization for query generation. In this section, we look at the corresponding problem for rule pairs i.e. given a pair of transformation rules (r_1, r_2) we need to generate a SQL query which can exercise *both* the rules when optimized. The rule patterns for individual rules can also be leveraged for generating necessary conditions (as in Section 3.1) to exercise a pair of rules by using the idea of rule pattern *composition.*

Consider the two rule patterns shown in Figure 3 for the join commutativity rule and the rule for pulling up the Group-By Aggregate over a join. In order to generate a query that can exercise *both* the rules, we can combine the rule patterns in the following ways: (1) Create a new pattern with a root operator as join or UNION and both the initial patterns as the corresponding children. (2) Substitute any generic operators in a pattern (represented as circles in the patterns in Figure 3) with the other pattern to create a composite pattern.

We have extended the query generation module to handle query generation for a pair of rules as follows. We compose the two rule patterns as described and generate a query corresponding to each of the composite patterns and pick the query with the *least number of operators* that exercises both the rules. Note that rule composition captures an important interaction between rules; rule r_1 is exercised on an expression which is an *input* to the expression on which rule r_2 is exercised. Of course, there are potentially other interesting patterns of rule interactions. We discuss other variants of the query generation problem in Section 7.

4. TEST SUITE COMPRESSION PROBLEM

One approach for testing rule correctness is to leverage *stochastic testing* (e.g. as in [1][11][17]). The idea is to generate a complex random query that exercises a given rule. We then: 1) Execute the original query. 2) Execute the plan obtained for the query with the rule turned off. 3) Check if the results of (1) and (2) are the same or not. In order to have sufficient confidence in the correctness of a rule, we may need to repeat the above validation step for several such randomly generated queries. Thus, for each transformation rule we need to validate its correctness for **k** distinct queries (where **k** is an input parameter that we refer to as the *test suite size*). Since the queries generated are potentially complex and need to be executed, the time taken to run these test suites can be significant. In this section, we formally present the problem of *test suite compression* (first described in Section 2.3), which can significantly improve the efficiency of correctness testing. We first show that this problem is NP-Hard. In Section 5, we present two algorithms for solving the test suite compression problem.

4.1 Problem Statement

Let $\mathcal{R} = \{r_1, \ldots r_n\}$ denote the set of transformation rules. Let the test suite size be **k**, and let TS denote the overall test suite for all rules (Section 2.3), i.e. $TS = \cup_i TS_i$. The relationship between the rules and the queries in the test suites can be represented by a bipartite graph (see Figure 4). An edge between a rule r_i and a query q_j denotes the fact that rule r_i is exercised when query q_j is optimized. Note that a query belonging to TS_i (the test suite generated for rule i_i) can potentially exercise other transformation rules as well. By exploiting this information, we can improve the efficiency of test suite execution, illustrated by the following example.

Example 1- Consider the case when $\mathcal{R} = \{r_1, r_2\}$. Let the rule test suite size (**k**) be 1, and the corresponding test suite for the rules are $TS_1 = \{q_1\}$ and $TS_2 = \{q_2\}$. Thus, $TS = \{q_1, q_2\}$. Assume that r_1 is the only rule triggered when q_1 is optimized, whereas both

rules are triggered when q_2 is optimized. Suppose the costs associated with the queries are as follows: Cost(q_1) = Cost(q_2) = 100. Cost(q_1, ¬{r_1}) = 180. Cost(q_2, ¬{r_2}) = 120. Cost(q_2, ¬{r_1}) = 120.

The BASELINE method for test suite execution (Section 2.3) would be as follows:

- Execute Plan(q_1) and Plan(q_1, ¬{r_1}).
- Execute Plan(q_2) and Plan(q_2, ¬{r_2}).

The cost of the BASELINE method for this example is: (100+180) + (100+120) = 500.

One alternative is to use query q_2 for validating both rules. The cost of this strategy would include:

- Execute Plan(q_2) and Plan(q_2, ¬{r_1})
- Execute Plan(q_2, ¬{r_2})

Note that we do not need to execute Plan(q_2) when validating r_2 since we have executed it when validating r_1 (and thus its results are already available) Thus, the cost of this strategy is (100+120) + (120) = 340, which is less expensive than the BASELINE method.

From the above example, we note that there are two important observations that can be leveraged in test suite compression. First, when a query *q* exercises multiple rules, Plan(*q*) (with all rules enabled) needs to be executed *only once.* Second, since the randomly generated queries can have widely varying costs, we can leverage this fact to reduce the cost by choosing queries with lower cost. Finally, the cost of the query with the *rules disabled* could be significantly higher than when the rules are enabled (e.g. if the rule is responsible for pushing selection below a join, disabling that rule can dramatically increase the cost of the query). Therefore, ideally this also needs to be factored in during test suite compression. We now formally define the test suite compression problem. We describe it for the singleton rule case, although the formulation extends in the obvious way for the case of rule pairs.

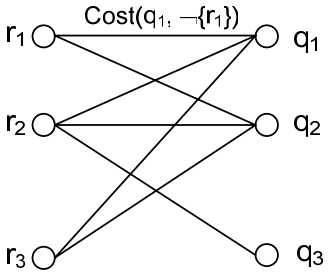

Figure 4. **Bipartite Graph Representation**

Test Suite Compression Problem: Consider the bipartite graph **G** = (**V**,**E**), where **V** = ($\mathcal{R}$ ∪ TS) and **E**, the set of edges in the graph, is defined as follows: add an edge between a rule $r_i \in \mathcal{R}$, and query $q \in$TS if optimizing *q* exercises rule r_i. Each node r_i in R is assigned a cost 0, and each node q ∈ TS is assigned a cost equal to Cost(*q*). For each edge (r_i, *q*) assign the cost equal to Cost(q, ¬{r_i}) i.e. the cost of executing the query *q* with the rule r_i disabled (see Figure 4). The test compression problem is to find a subgraph G' = (V',E') of the above bipartite graph such that:

1) V' = ($\mathcal{R}$ ∪ TS') and TS' ⊆ TS. In other words, the subgraph contains all rules from G and a subset of the queries from G.
2) Each node r ∈ $\mathcal{R}$ in G' has degree equal to the test suite size **k**.
3) The sum of the edge and node costs in G' is minimized.

Intuitively, we intend to find the mapping of queries to rules with the minimum cost (condition 3) such that every rule is accounted for (condition 1) while ensuring that each rule is mapped to exactly **k** queries which is the size of the test suite (condition 2). Note that any subgraph of the bipartite graph that satisfies property 1) and 2) is a valid test suite. The node cost for the query nodes is used to model the fact that for queries shared between multiple rules, the original Plan(*q*) needs to be executed only once. The execution of the test-suite would proceed as follows. For each q ∈ TS', we execute Plan(q) once. For each edge (*q*, *r*), we execute the Plan(*q*, ¬{*r*}) and compare the results with those obtained from Plan(*q*). Thus, the sum of the edge and node costs is equal to the cost of executing the test suite. Since the out-degree of each node r ∈ $\mathcal{R}$ in the subgraph is known to be **k**, we are guaranteed to execute **k** distinct queries for each rule in the set $\mathcal{R}$.

4.2 Hardness

Claim: The Test Suite Compression Problem is NP-Hard.

Proof: See Appendix A for the proof. We show hardness by reducing an arbitrary instance of the Set Cover problem [10] to a simplified version of the Test Suite Compression (TSC) problem.

5. ALGORITHMS FOR TEST SUITE COMPRESSION PROBLEM

In Section 4, we introduced the Test Suite Compression (TSC) problem, and why it can be important in significantly improving the efficiency of correctness testing of rules. We also showed that TSC is computationally hard. In this section, we present two algorithms for this problem. In Section 6, we study the effectiveness of these two algorithms via an empirical evaluation, and compare them with the BASELINE method (described in Section 2.3).

5.1 Applying the Set Cover Heuristic

In section 4.2 we showed that the test suite compression problem is NP-Hard. The reduction (see Appendix A for details) demonstrated that the Set Cover problem is isomorphic to a simplified version of the test suite compression problem (that uses a test suite size **k** =1). Since good approximation algorithms exist for the set cover problem [19], a natural question is whether such an algorithm can be leveraged for the test suite compression problem.

Observe that the simplified version of the test suite compression problem (that was shown to be isomorphic to the set cover problem) uses a test suite size **k** = 1. To incorporate this parameter we therefore adapt an algorithm for the corresponding general version of the set cover problem, which is called the *Constrained Set Multicover problem* [19]. The constrained set multicover problem takes as input a set **U,** a number m_e for each $e \in$ **U**, and a set of subsets **S** of elements in **U**. Each $s \in$ S has a cost C(*s*). The goal is to find the subset with the minimal cost such that:

- Each element *e* of U is covered m_e times
- Each $s \in$ **S** can be picked at most once

The test suite compression problem can be mapped to it as follows. The set of rules $\mathcal{R}$ maps to input set **U**. For each query q, we map RuleSet(q) to the corresponding subset $s \in \mathbf{S}$, the node cost Cost(q) to the corresponding C(s). For all rules r $\in \mathcal{R}$, we set m_r to **k**. Figure 5 shows how we can adapt the greedy algorithm for the Set Multicover problem [19] to compute the set of queries with minimal cost such that each rule is mapped to exactly **k** distinct queries in the test suite. The algorithm collects queries to be picked in the set TS'. It tracks the set of rules that have already been covered in R'. In Step 2 we check if the set of rules that have been covered in R' is complete. Note that this step includes the check that each rule in $\mathcal{R}$ has been covered **k** times. In Step 3, we compute the "benefit" of each query that has not been picked. We define a rule to be *remaining* if it exercised by less than **k** of the queries already picked. The benefit of a query is defined as the number of remaining rules that are covered normalized by the cost of the query. For example consider a query q with RuleSet(q) = $\{r_1, r_2, r_3\}$. Let the test suite size **k** be set to 2. Assume that the rule r_1 has already been covered by 2 queries in the set TS'. The remaining rules for query q are thus $\{r_2, r_3\}$. The greedy algorithm picks at any point the query with the highest "benefit" (Step 4).

Input: Bipartite Graph G = (($\mathcal{R} \cup$ TS), E), Test Suite Size **k**

Output: A bipartite graph G' = (($\mathcal{R} \cup$ TS'), E'), a subgraph of G with outdegree of each node in $\mathcal{R}$ = **k**

1. TS' = {}, R' = {}, E' = {}
2. **While** (R' != $\mathcal{R}$) **Do**
3. For each $q \in$(TS – TS'), compute Benefit(q) = number of remaining rules covered / Cost(q)
4. Pick $q \in$(Q – Q') with the largest Benefit value
5. TS' = TS' $\cup$ {q}; R' = R' $\cup$ RuleSet(q);
6. Add edges corresponding to q and the remaining rules it covers, to E'.
7. **End While**
8. Return G' = (($\mathcal{R} \cup$ TS'),E')

Figure 5. **Greedy Algorithm based on the Set MultiCover problem**.

Consider Example 1 (see Section 4.1) where query q_1 exercises the rule $\{r_1\}$ and q_2 exercises the rules $\{r_1, r_2\}$. Since Cost(q_1) = Cost(q_2), q_2 has the higher benefit and as a result the greedy algorithm in Figure 5 would find the optimal solution for the example. In general, however this algorithm tries to minimize the total cost of the query nodes in the subgraph and *does not* model the edge costs (see Section 4) which accounted for the costs of executing queries with the corresponding rules disabled. Since, in general, the edge costs could be potentially significant, we now present another algorithm that takes into account the edge costs.

5.2 A Constant Factor Approximation Algorithm

In this section, we present a heuristic for the test suite compression problem that takes into account the edge costs. We also show that our algorithm is a factor 2 approximation of the optimal solution to the test suite compression problem. Intuitively, the algorithm selects for each rule, the **k** queries with the lowest cost with that rule disabled (i.e. edge cost). Unlike the *SetMultiCover* algorithm (Section 5.1), this algorithm assumes independence between the rules, thereby ignoring the benefits obtained from potentially sharing queries between the test suites of different rules (see Example 1).

The algorithm (we refer to it as *TopKIndependent*) is shown in Figure 6. For each rule r in the set $\mathcal{R}$, we first obtain the set of all queries in TS that exercise rule r. (Step 4). The loop (lines 5-11) picks the **k** queries with the lowest edge cost, i.e. the cost of the query when the rule r is disabled. This step is repeated for all the rules in the set $\mathcal{R}$.

Referring once again to Example 1 (Section 4) where q_1 exercises $\{r_1\}$ and q_2 exercises the rules $\{r_1, r_2\}$. Since **k** is 1, the *TopKIndependent* algorithm would choose the query that has the minimum edge cost for each rule. For both the rules, q_2 has the smaller edge cost when compared to q_1. Thus, for Example 1 the algorithm in Figure 6 would also find the optimal solution.

Input: Bipartite Graph G = (($\mathcal{R} \cup$ TS), E),Test Suite Size **k**

Output: A bipartite graph G' = (($\mathcal{R} \cup$ TS'), E'), a subgraph of G with outdegree of each node in $\mathcal{R}$ = **k**

1. TS' = {}, R' = {}, E' = {}
2. **For** each rule r in $\mathcal{R}$ **Do**
3. count = 0
4. Let W = Subset of queries in TS that includes rule r in its RuleSet
5. **While** (count < k) **Do**
6. Pick $q \in$ W with minimal value Cost(q, ¬{r}) value
7. W = W - {q}
9. TS' = TS' $\cup$ {q}; count = count+1
10. Add edge corresponding to (r,q) to E'
11. **End While**
12. R' = R' $\cup$ RuleSet(q);
13. **End For**
14. Return G' = (($\mathcal{R} \cup$ TS'),E')

Figure 6. *TopKIndependent* Algorithm

Although the *TopKIndependent* algorithm ignores query node costs, we can show that it provides a solution that is guaranteed to be within a factor 2 of the optimal solution. Note that the *SetMultiCover* algorithm, on the other hand does not provide a constant factor approximation.

Claim: *TopKIndependent* algorithm is a factor 2 approximation algorithm for the test suite compression problem.

Proof: Consider any rule $r \in \mathcal{R}$. For rule r, the *TopKIndependent* algorithm chooses the **k** queries with the least expensive edge costs (i.e. cost of query with rule r disabled). Let us denote this set of queries by TS(r). Note that the *maximum cost* of any solution obtained by *TopKIndependent* over all rules in $\mathcal{R}$ cannot exceed:

$$MaxCost = \sum_{r \in R} \sum_{q \in TS(r)} Cost(q) + Cost(q, \neg\{r\})$$

The above (upper bound) occurs when for each rule, there is no query chosen for that rule which is shared with any other rule in $\mathcal{R}$.

Now we present a lower bound for *any* solution to the test suite compression problem. This lower bound occurs when: (a) The **k** cheapest queries (in terms of edge costs) are chosen for each rule r, and (b) For each rule, all the queries picked for the rule r are shared with some other rule.

$$MinCost = \sum_{r \in R} \sum_{q \in TS(r)} Cost(q, \neg\{r\})$$

Observe that the above cost does not correspond to any valid solution since it ignores Cost(q) entirely. However, it is a valid lower bound on the cost of any solution.

Notice that for any rule r and query q, $Cost(q) \leq Cost(q, \neg\{r\})$ since for a well behaved optimizer disabling a rule can only increase the cost of the resulting plan. This is because when a rule is disabled, one of the following two possibilities can occur: (1) It may not impact the plans considered by the optimizer, in which case the resulting plan (and hence the cost) is the same, or (2) It can reduce the number of plans considered by the optimizer, in which case the cost of the plan chosen can only be higher.

Now consider the ratio f = (MaxCost / MinCost). Since $Cost(q) \leq Cost(q, \neg\{r\})$, $f \leq 2$. Note that f=2 occurs when Cost(q) = Cost(q,$\neg${r}). Since the optimal solution has a cost higher than MinCost, and actual cost of the solution picked by *TopKIndependent* is no higher than MaxCost, we know that the solution picked by *TopKIndependent* has a cost that is within a factor of 2 of the optimal solution.

5.3 Extensions for Rule Pairs

In this section, we discuss how the algorithms we described in Sections 5.1 and 5.2 for the test suite compression problem can be extended for testing pair-wise rule interactions.

The test suite compression problem for testing rule pairs is very similar to the original formulation, with the key difference being that the input is a set of *rule pairs* rather than a set of individual rules. We denote the set of all rule pairs by $\mathcal{P}$, i.e. $\mathcal{P}$ = {{r_1, r_2}, {r_1, r_3}, …{r_{n-1}, r_n}}. Thus, for each element $p \in \mathcal{P}$, we need to find the mapping of **k** distinct queries such that the cost of executing the test suite is minimized. An example of the bipartite graph corresponding to the test suite compression problem for rule pairs is shown in Figure 7. Note that we add an edge between a rule node and a query node (q) only if *both* the corresponding rules are exercised when query q was optimized. In the example bipartite graph shown in Figure 7, all the queries exercise both rule pairs. For an edge between a rule node $\{r_i, r_j\}$ and a query q, the edge cost is the cost of executing q with *both* rules disabled i.e. Cost(q, $\neg\{r_i,r_j\}$). Thus the cost of executing q_1 when both r_1 and r_2 are disabled is 150.

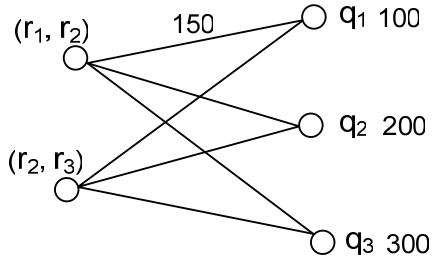

Figure 7. Bipartite Graph for Rule Pairs

The algorithms presented in Section 5 extend in a straightforward manner for the case of rule pairs. However, the main difference is that the cost of *creating* the bipartite graph which is an input to both algorithms itself can become significant. Consider a query q that exercises five rules. In the original formulation of the problem, we need to add five edges to the corresponding rule nodes. For the case of pairs, we need to add $^5C_2$ edges corresponding to all the pairs of the rules. Recall that the edge costs model the cost of turning off a particular pair of rules. In general for a query that exercises n rules, we need $^nC_2$ invocations of the query optimizer to compute the edge costs corresponding to that query node. For complex queries, the number of rules exercised could potentially be high and thus the cost of building the initial bipartite graph could be significantly higher when compared to the case of singleton rules. Thus, scalability of the algorithms with number of rules becomes a significant issue.

5.3.1 Exploiting Monotonicity to Reduce Optimizer Invocations

We observe that the increase in the cost of constructing the bipartite graph only affects the *TopKIndependent* algorithm (Section 5.2). This is the because the *SetMultiCover* algorithm described in Section 5.1 does not model the edge costs and uses only the query node costs as a basis for picking nodes.

For the *TopKIndependent* algorithm, we present a technique that can help reduce the number of edges for which the cost needs to be determined by the algorithm (in Steps 5-10 of Figure 6). Our idea is to exploit the observation that for any query Cost(q) ≤ Cost(q, ¬R), where R is any subset of rules in RuleSet(q). Suppose for a rule pair $p \in \mathcal{P}$ the queries that exercise the rule pair is TS(p). Our goal is to find the **k** edges (with lowest cost) from p to queries in TS(p). We sort queries in TS(p) in increasing order of the original node cost, i.e. by Cost(q). We also maintain a priority queue of size **k** of edges for which we have computed the actual cost (by invoking the optimizer). Initially, the priority queue is empty. Each time we consider the next edge (say e) from TS(p) we check if the node cost corresponding to that edge has a higher value than the edge with the **k***th* highest cost in the priority queue. If so, we can terminate since we know that the cost of the edge e (and all remaining edges for p) must have a higher cost. If not, we compute the actual cost and add the edge to the priority queue (potentially evicting an existing edge with the highest cost to maintain the size of k in the priority queue). Our experimental results (Section 6) indicate that this optimization can significantly reduce the costs of creating the bipartite graph for testing rule pairs.

The following example illustrates the above optimization. Consider the bipartite graph shown in Figure 7 and let the test suite size k be 1. For the node $\{r_1, r_2\}$, we need to find the edge with minimum cost. We illustrate how we can potentially achieve this *without* having to compute all the edge costs. Consider the set of all queries with edges to $\{r_1, r_2\}$. In our case this is $\{q_1, q_2, q_3\}$. We order the queries in the increasing order of node costs. We first compute the edge cost for query q_1, i.e. (Cost (q_1, $\neg\{r_1, r_2\}$) by invoking the query optimizer. As shown in the figure, suppose this edge cost is 150 units. Note that the original query cost of q_2(200) and q_3(300) are higher than this value. This implies that the corresponding edge costs can only be higher (since disabling a pair of rules can only increase the cost of the resulting plan). Thus, we can stop enumerating the edges at this point and return the current edge (corresponding to q_1) as the minimal cost edge.

In this paper, we have primarily focused on testing single transformation rules and the interactions of rule pairs because these are the most common scenarios for testing transformation rules. We discuss interesting extensions to study as part of future work in Section 7.

5.4 Summary

In this section, we presented two algorithms for solving the test suite compression problem. We first showed how to leverage the greedy heuristic for the *SetMultiCover* problem (which ignores the edge costs). Next we presented the *TopKIndependent* algorithm that ignored the benefits of using the same query for different rules (i.e. the query node costs). However, this algorithm guarantees a constant factor approximation to the optimal solution. We have implemented both these algorithms, and we describe results of our experimental comparison in Section 6.

6. EXPERIMENTS

We have prototyped the framework described in this paper (see Figure 2) on Microsoft SQL Server. In this section, we present the results of our experiments for evaluating the effectiveness of the techniques presented in this paper. In particular, the goals of our experiments are:

- Compare the efficiency of query generation using rule patterns (Section 3) with the randomized query generation approach. We do this for singleton rules as well as rule pairs.
- Compare the effectiveness of the *SetMultiCover* algorithm (Section 5.1) and *TopKIndependent* algorithm (Section 5.2) for the Test Suite Compression problem for both singleton rules as well as rule pairs.
- Study the importance of exploiting monotonicity (Section 5.3.1)

6.1 Databases

As described in Section 2, we are given as input a test database. For our experiments, we use tables from the TPC-H [21] database. We focus on *logical* transformation rules (see Section 2.1) in this evaluation, these rules are by and large exercised regardless of the data size or distribution. We use a set of around 30 logical transformation rules of the optimizer (that cover the most commonly used operators including selections, joins, outer joins, semi-joins, group-by etc.). We have also evaluated our tests on other databases with different schemas and sizes, and the results are similar to those presented below, so we do not report those results here. We defer a more thorough evaluation (that includes implementation rules) to future work.

6.2 Results

6.2.1 Leveraging Rule Patterns for Query Generation

In our first experiment, we study the impact of rule patterns for query generation (see Section 3). We report our results both for singleton rules as well as rule pairs.

Figure 8 shows the number of trials required to generate a query for each singleton rule. Observe that our technique of pattern based generation of logical query trees (Section 3) is able to generate a query that exercises the given rule in a very small number of iterations (typically 1 or 2, and never more than 4). This is a significant improvement relative to random query generation, where generating for certain rules it takes close to 40 attempts before a query which exercises that rule can be generated. For the entire set of 30 rules, the total number of trials for RANDOM is 234 whereas for PATTERN it is 38.

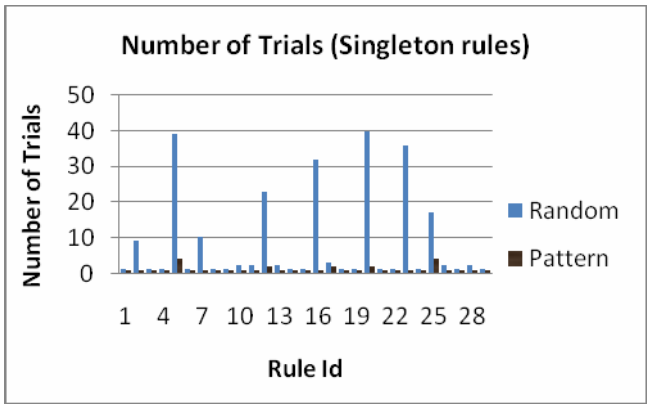

Figure 8. Random vs. Pattern based generation for singleton rules.

The difference in efficiency between RANDOM and PATTERN is even more significant for rule pairs, as can be seen in Figure 9. Note that the y-axis uses a logarithmic scale. We show the results when the number of rules (n) = 15, 30, and therefore number of rule pairs = $^{15}\mathbf{C}_2$ and $^{30}\mathbf{C}_2$ respectively. For n=15, RANDOM requires 1187 trials, whereas PATTERN requires only 383 trials. For n=30, PATTERN shows a 13x improvement (RANDOM requires over 13,000 trials whereas PATTERN requires less than 1,000 trials). This is because in general, the chance that a query generated using a random generation procedure exercises a *set* of rules drops rapidly as the cardinality of that set increases. For example, we observe rule pairs for which it takes close to 100 trials to generate a valid query. On the other hand, with PATTERN, most queries were again generated within 1 or 2 trials, and the maximum number of trials we observed for a rule pair was 5.

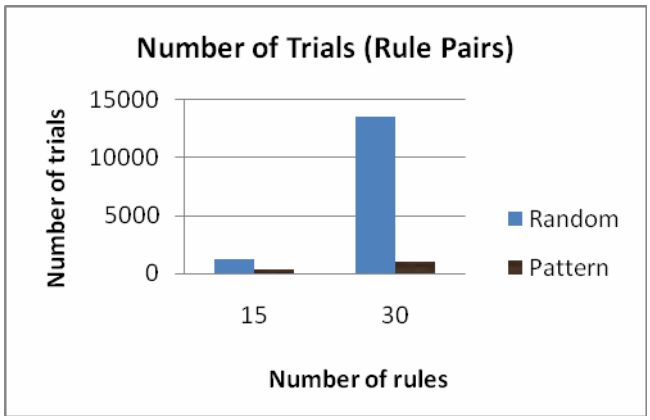

Figure 9. Random vs. Pattern based generation for rule pairs.

Figure 10 shows the time required to generate the queries for the same data points as in Figure 9 (once again the y-axis uses a log scale). This figure shows that the efficiency of PATTERN over RANDOM in number of trials also extends directly to a reduction in the time required to generate the test cases.

Together, these results clearly show the importance of rule pattern based query generation for singleton rules as well as rule pairs.

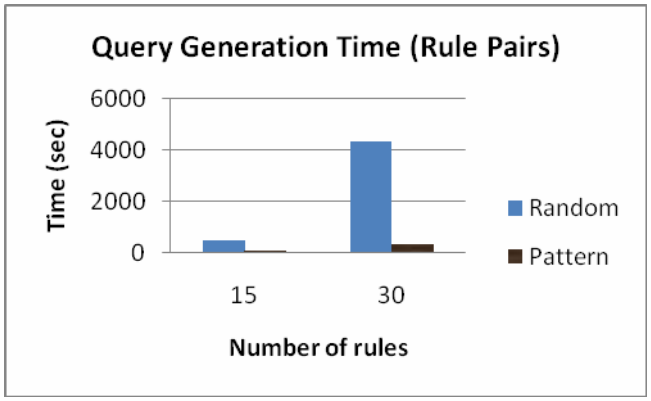

Figure 10. Random vs. Pattern based generation for rule pairs.

6.2.2 Test Suite Compression

Recall that the test suite compression problem (presented in Section 4) is important for the efficiency of correctness testing of rules. In this experiment we compare the three approaches for the test suite compression problem: BASELINE (Section 2.2); *SetMultiCover* (Section 5.1), which we refer to as SMC in the graphs; and *TopKIndependent* (Section 5.2), which we refer to as TOPK in the graphs (we use a test suite size **k** of 10). We show results for singleton rule as well as rule pairs. Note that for all graphs in this section, *we use a log scale for the y-axis,* which represents the total cost (we use the optimizer estimated cost) of the solution, as the number of rules is varied.

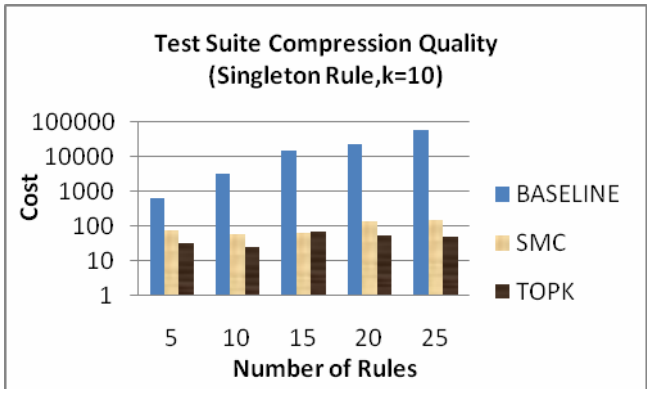

Figure 11. Test suite compression for singleton rules.

For singleton rules (see Figure 11), we observe that both SMC and TOPK obtain solutions that are significantly better than BASELINE (anywhere between one and three orders of magnitude). This shows that unlike BASELINE, both SMC and TOPK are able to take advantage of using a single query for validating multiple rules.

For the case of rule pairs (see Figure 12) however, the results are somewhat different. While TOPK continues to produce the lowest cost solutions, SMC's solution vary between good to significantly worse than BASELINE. The reason for this is that SMC does not take into account the edge costs (i.e., the cost of a query when a set of rules is disabled). Therefore in certain cases, it selects queries that have low cost when optimized with *all* rules, but whose cost is significantly higher when certain rule pairs are turned off. We observe that although this phenomenon occurs even in the singleton rule case, it is much more pronounced in the case of rule pairs since there are many more opportunities for such queries to arise. On the other hand since TOPK always picks the **k** edges with lowest cost, it is more robust to this problem when compared to SMC.

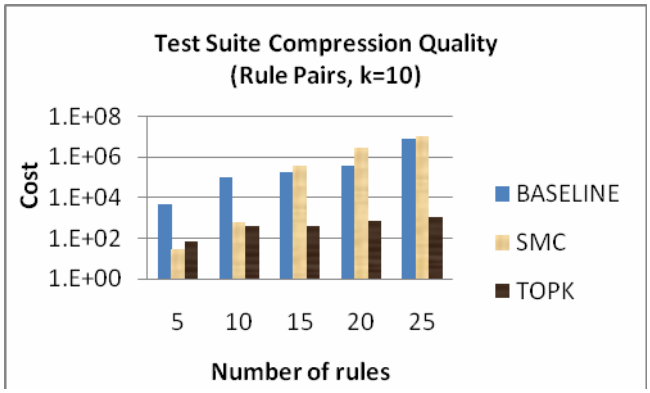

Figure 12. Test suite compression for rule pairs.

In our next experiment, we fix the number of rules n=15 (and thus the number of rule pairs to $^{15}C_2$), and vary **k**, the test suite size. The result of this experiment is shown in Figure 13. We see that TOPK is again the best algorithm across all values of **k**. We note that for very low values of **k**, (e.g. **k**=1) SMC produces good solutions, but at larger values of k, its quality drops significantly. Once again, this is due to the fact that as k increases, it becomes more likely to find queries where turning off a rule pair causes the cost to rise sharply (this cost is ignored by SMC).

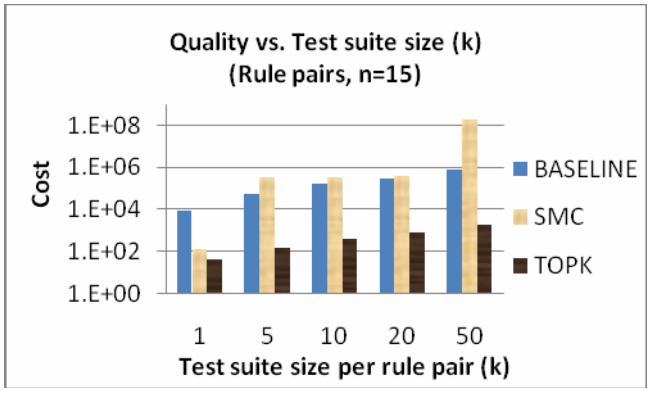

Figure 13. Impact of varying the test suit size on quality of solution.

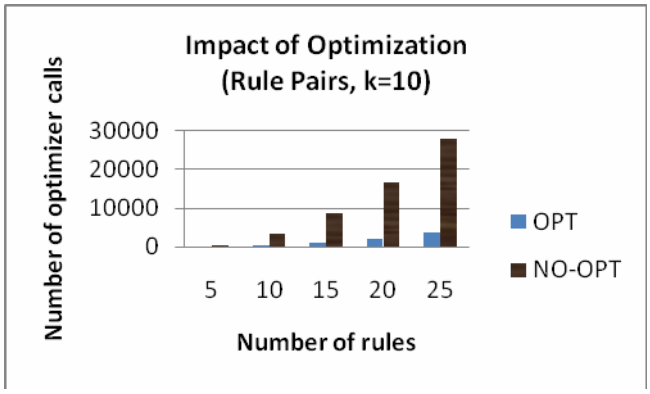

Figure 14. Exploiting Monotonicity

In our final experiment, we measure the importance of exploiting monotonicity for the TOPK algorithm (see Section 5.3.1). Recall that this optimization can potentially save the algorithm from making a large number of invocations of the query optimizer to compute the edge costs. Figure 14 shows the impact of using this optimization for the case of rule pairs. We see that exploiting monotonicity saves between a factor of 6x to 9x of the optimizer calls without affecting the actual quality of the result (i.e., it is a sound technique).

From these experiments we see that TOPK is consistently the best approach. When coupled with the observation that TOPK is guaranteed to be a factor 2 approximation (Section 5.2) of the optimal solution, we conclude that TOPK is an attractive approach for correctness testing.

7. DISCUSSION

In this paper we outlined a framework for testing transformation rules. We presented techniques for query generation (Section 3) for evaluating code coverage of rules as well as techniques for optimizing the execution of test suites (Section 5) for validating the correctness of rules. In this section, we discuss some interesting extensions and potential directions for future work for rule testing.

Variants of Query Generation Problem: The query generation problem as studied in this paper was as follows: Given a rule (or a pair of rules), generate a SQL query such that the rule (or rule pair) is exercised. While tracking if a rule is exercised is indeed important, note that a rule that is exercised may not influence the final plan choice of the optimizer. Intuitively, a rule is *relevant* for a query if turning off the rule results in the optimizer picking a different plan. It is interesting to study the following variant of the query generation problem: Given a rule, generate a SQL query such that the rule is relevant for the query.

In Section 3.2, we looked at rule composition for rule pairs which captures an important interaction between rules; rule r_1 is exercised on an expression which is an *input* to the expression on which rule r_2 is exercised. Similarly, there are other potential definitions of rule interactions, for example a rule r_2 is exercised on an expression which was obtained as a result of exercising rule r_1. We intend to extend the query generation techniques to handle such variants as part of future work.

In this paper we assume that we are given as input a database, i.e., the database is fixed. While this assumption is reasonable for a large class of transformation rules, there are certain rules whose exercising is dependent on the properties of the schema as well as the database instance. For example, consider a transformation rule that optimizes star join queries. For this rule to be exercised, certain foreign key constraints must be defined in the schema. In order to capture such rules, it may be necessary to augment our query generation module to modify the schema and/or the database instance.

Variants of Test-Suite Compression Problem: We introduced the test-suite compression problem in Section 4.1. Given the original test-suite, the problem is to find the least-cost mapping of queries to rules while preserving the invariant that each rule is mapped to **k** distinct queries. Note that in this version, we can potentially reuse queries for validating different rules. A stronger invariant is one that still preserves all the distinct queries in the original test suite (i.e. there is no sharing of queries across rules). The corresponding problem then is to find the least-cost mapping of queries to rules such that each query in the original test suite is mapped to *exactly* one rule. We can show that this problem reduces to bipartite matching and thus can be solved efficiently; we omit details due to lack of space.

8. RELATED WORK

The architecture of a transformation rule based framework for query optimization is described in detail in [12]. Transformation rules have been adopted in many commercial systems including Tandem's Non-Stop SQL [7], IBM DB2 [16] and Microsoft SQL Server [13].The authors in [11] present an overview of issues related to testing a commercial query optimizer. One of the earliest papers to talk about query optimizer testing is [18]; it focuses on tools that can help generate data having certain characteristics (such as correlations between attributes) as well as generate queries whose join graph has certain properties.

A stochastic testing scheme for SQL is described in [17]. They present a tool (RAGS) in order to generate random complex SQL queries that are valid and use it for stress testing the SQL parser and to check if the results of these queries are the same across different database systems. The work in [1] extends the random query generator using genetic algorithms to ensure certain properties of the generated queries (such as nonempty results). There has been some recent work on the problem of query generation [6][5][15], this work is primarily concerned with generating queries and database instances such that certain cardinality constraints (e.g. the cardinality of a particular join result is 1000) are satisfied. None of the previous work on query generation has been concerned with generating queries such that certain transformation rules are exercised.

The work in [9] describes an interface that can generate a SQL tree from a logical expression (corresponding to the Generate SQL component in Figure 2). It is largely to facilitate the manual creation of unit tests, where the developer can design a particular logical tree and then use the techniques in [9] to generate the corresponding SQL. We note that the techniques in [9] do not address how to generate an appropriate logical tree for generating queries that exercise certain rules. While we use a similar component as part of our query generation module, our idea of leveraging rule patterns from the optimizer engine and rule composition is the key difference that enables us to efficiently generate queries that exercise a particular rule (or rule pair).

There has been previous work on the problem of efficiently executing test runs for database applications [14] which also study algorithms to minimize the total time to execute the tests. For the problem setting in [14], the key point was carefully factoring the cost of resetting the state of the database, which is not relevant for the problem of executing test suites that is studied in this paper. Finally, while there has been previous work on compressing workloads [8][20], this has been in the context of physical database design. The aim is to obtain a subset of queries from the original workload such that the physical database design for the compressed workload is similar to the physical database design that would have been generated for the original workload. Since the compression techniques are used are specific to the problem of physical database design, they are not applicable to our problem of test suite compression.

9. CONCLUSION

Transformation rules play a crucial part in the ability of modern query optimizers to find a good query execution plan. Despite

their obvious importance there has been relatively little work focused on testing rule based query optimizers. In this paper, we present an initial framework for testing transformation rules. We show how to leverage rule patterns for efficient query generation to generate test cases that exercise specific rules. We also introduce the novel problem of test suite compression and present a solution that can significantly reduce the time required for correctness testing of these rules. As part of future work we intend to extend our framework to efficiently test rule interactions beyond rule pairs and also examine issues in testing other components of the query optimizer.

10. REFERENCES

[1] H. Bati, L. Giakoumakis, S.Herbert, A.Surna. A genetic approach for random testing of database systems. *Proceedings of VLDB 2007*.

[2] B.Beizer. Software Testing Techniques (2nd ed.) Van Nostrand Reindhold Co.1990.

[3] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. *In Proceedings of PODS 1998*.

[4] K.Billings. A TPC-D Model for Database Query Optimization in Cascades. Ms. Thesis. Portland State University. 1996

[5] C.Binning, D.Kossman, E.Lo, T.Ozsu. QAGen: Generating Query-Aware Test Databases. *Proceedings of ACM SIGMOD 2007*.

[6] N.Bruno, S.Chaudhuri, D.Thomas. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE TKDE 18(12) 2006*.

[7] P.Celis. The Query Optimizer in Tandem's new ServerWare SQL Product. *Proceeding of VLDB 1996*.

[8] S.Chaudhuri, A.Gupta, V.Narasayya. Compressing SQL Workloads. *Proceedings of SIGMOD 2002*.

[9] M.Elhemali, L.Giakoumakis. Unit Testing Query Transformation Rules. *Proceedings of DBTest Workshop 2008*.

[10] M.R. Garey, and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[11] L.Giakoumakis, C. Galindo-Legaria. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. IEEE Data Engineering Bulletin 2008 vol. 31 (1).

[12] G.Graefe, W.McKenna. The Volcano Optimizer Generator. Extensibility and Efficient Search. *Proceeding of ICDE 1993*.

[13] G.Graefe. The Cascades Framework for Query Optimization. Data Engineering Bulletin, 18(3), 1995.

[14] F.Haftmann, D.Kossmann, A.Kreutz. Efficient Regression Tests for Database Applications. *Proceedings of CIDR 2005*.

[15] C.Mishra, N.Koudas, C.Zuzarte. Generating Targeted Queries for Database Testing. *Proceedings of SIGMOD 2008*.

[16] H.Pirahesh, J.Hellerstein, W.Hasan. Extensible Rule Based Query Rewrite Optimization in Starburst. *Proceedings of SIGMOD 1992*.

[17] D.Slutz, Massive Stochastic Testing of SQL. *Proceeding of VLDB 1998*.

[18] M.Stillger, J.C.Freytag. Testing the quality of a query optimizer. Data Engineering Bulletin, 18(3), 1995.

[19] V.Vazirani. Approximation Algorithms. Springer-Verlag 2003.

[20] D.Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. *Proceedings of VLDB 2004*.

[21] TPC Benchmark H. Decision Support. http://www.tpc.org

APPENDIX A

Claim: The Test Suite Compression Problem (Section 4.1) is NP-Hard.

Proof: We show hardness by reducing an arbitrary instance of the Set Cover problem [10] to a simplified version of the Test Suite Compression (TSC) problem. Consider a simplified version of the TSC problem (we refer to it as S-TSC) where the edge and the query node weights are assigned the same unit weight and the test suite size **k** is 1. Observe that any valid solution for S-TSC has the following characteristics. Since we require every rule node to be part of the output subgraph, and **k**=1, there are exactly $|\mathcal{R}|$ edges in the subgraph, and hence total cost of the edges is $|\mathcal{R}|$. Thus, finding the lowest cost solution in S-TSC is equivalent to minimizing |TS'|, i.e. finding the smallest subset of queries in TS that results in exercising each rule.

We now reduce the Set Cover problem to S-TSC. The Set Cover problem takes as input a set **U** and a set **S** of subsets of **U**. The goal is to find the smallest subset of **S** that covers all the elements in **U**. We can map an arbitrary instance of the Set Cover problem to the S-TSC problem as follows. The set **U** maps to the set of rules $\mathcal{R}$. Thus, each element $s \in \mathbf{S}$ maps to a subset of the rules. Recall that RuleSet(q) (Section 2.2) denotes the set of rules that are exercised when query q is optimized. For each $s \in \mathbf{S}$, we generate a corresponding query q such that RuleSet(q) = s. This can be done as follows. Note that for each individual rule r, we can generate a query that exercises rule r (e.g. using the method described in Section 3). Therefore, we can then generate one query expression for each rule in s and construct a single query that is a UNION of each of the individual query expressions (including additional NULL columns to make the expressions union-compatible if required). Observe that by this construction the UNION query that is guaranteed to exercise all the rules in s. The set of query nodes in TS is the union of all queries generated in this manner. We add edges between a query q in TS and the nodes corresponding to RuleSet(q) in $\mathcal{R}$.

From the above construction, it is easy to see that Set Cover for U, S) is isomorphic to the S-TSC problem. Thus S-TSC is NP-Hard. Since S-TSC is a strict simplification of TSC, we conclude that the TSC problem is NP-Hard.