Minimizing Database Repros using Language Grammars

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

ABSTRACT

Database engines and database-centric applications have become complex software systems. Ensuring bug-free database services is therefore a very difficult task. Whenever possible, bugs that are uncovered during testing are associated with a *repro*, or sequence of steps that deterministically reproduce the problem. Unfortunately, due to factors such as automated test generation, repros are generally too long and complex. This issue prevents developers reacting quickly to new bugs, since usually a long manual "repro-minimization" phase occurs before the actual debugging takes place. In this paper we present a fully automated technique to minimize database repros that leverages underlying language grammars and thus is significantly more focused than previous approaches. Our approach has been successfully used in two commercial database products to isolate and simplify bugs during early development stages. We show that our technique consistently results in repros that are as concise or simpler and obtained much faster than alternative ones carefully constructed manually.

1. INTRODUCTION

Building database systems and database-centric applications is a complex task along many dimensions. Modern relational database systems (or DBMSs) have a very rich set of features, support a rather complex query language, and constantly evolve to meet new requirements. Applications that are built on top of a DBMS further increase the overall system complexity, and demand steep engineering resources to operate without interruptions.

Reaching the goal of bug-free data management services usually involves the related activities of *testing* and *debugging*. Testing typically uncovers the presence of a "bug" (or undesirable system behavior), and debugging is used to identify the root cause of the problem and helps providing ways to fix it. When possible, each bug is associated with a precise set of steps, or *repro*, that deterministically reproduces the error. Too often, the starting point for the debugging process is a large repro with several aspects that are irrelevant to reproducing the bug. This is a consequence of either automatic randomized test generators (e.g., [9]), or real-world application scenarios.

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00

Figure 1(a) shows an instance of an MDX query (a query language for OLAP databases [7]) that generated an exception on an early build of Microsoft Analysis Services (a component of Microsoft SQL Server that includes OLAP and Data Mining capabilities [6])[1]. Although this example is not extremely complex, it might not be apparent, from inspecting the MDX query, what might be causing the server crash. This usually leads developers into a long and tedious process of *repro minimization*, in which the initial repro is made as small as possible while at the same time still reproducing the error. The rationale is that the shorter and more concise a repro, the more likely it is to understand the root cause of the problem and effectively fix it. Conceptually, we try to obtain a *min-repro*, i.e., the "simplest" version of the original repro that still reproduces the original problem. Further simplifying a min-repro would make the problem not reproduce any longer.

To motivate the techniques in this paper, Figure 1(b) shows the min-repro identified by our approach when applied to the original MDX query of Figure 1(a). Our techniques produced the min-repro in a few seconds, compared to almost an hour of work by an experienced developer to manually reach the same expression. Furthermore, the underlined portions of the min-repro in the figure correspond to crucial query fragments that, if removed, would produce a valid query that no longer reproduces the problem. Specifically, the problem reproduces whenever there is a nested `SELECT` clause (fragment (2) in the figure), a `DrilldownMember` function with a parameter that uses `NameToSet` in a member function (fragment (1) in the figure), and a specific projection in the 0 axis (fragment (3) in the figure). Removing any of these elements from the min-repro would make the problem cease to manifest. In summary, the information in Figure 1(b) provides a much better starting point for a debugging session than that of the original repro in Figure 1(a).

1.1 Problem Statement

In the context of this paper, a *repro* is a script in a database language (the example in Figure 1 uses the MDX language, but SQL is another –more popular– alternative). Note that we are not restricted to single queries, but instead a repro consists, in general, of a full workload. A minimization problem is associated with a *testing function* $\mathcal{T}$: *repro* $\rightarrow \{\checkmark, \times, \circledS, ?\}$, which determines whether the bug manifests for a given repro[2]. The semantics of the testing function $\mathcal{T}$ are as follows. $\mathcal{T}(r) = \times$ means that the repro r fails the test (and therefore the bug is reproduced for r). In turn,

[1]Our work originated in the context of MDX queries, but all the ideas are equally applicable to SQL, as we show in subsequent sections.

[2]In this paper we assume that $\mathcal{T}$ is deterministic. There are scenarios, however, for which this assumption does not hold (e.g., those resulting from race conditions). These scenarios are outside the scope of this paper.

```
WITH SET mySet0 =
       { [Employee].[Employee Department].[Department Name].
            &[Purchasing].&[Purchasing Manager].&[279] :
          [Employee].[Employee Department].[Department Name].
            &[Production].&[Production Technician].&[123] }
       AS SetAlias
     SET mySet1 AS DrilldownMember (
         NameToSet ( '[SalesTerritory].[SalesTerritory].
                       [Group].&[North America]' ),
         NameToSet ( '[SalesTerritory].[SalesTerritory].
                       [Group].&[Europe]' )
                    )
     SET mySet2 AS [Customer].[Customer Geography].[Country].
            &[Germany].Children
     MEMBER [Measures].[C0] AS mySet2.Count
     MEMBER [Measures].[C1] AS mySet0.Count
     MEMBER [Measures].[C2] AS mySet1.Count
SELECT mySet1 ON 0,
       { [Product].[Product Model Lines].[Model].
             &[Front Derailleur] :
          [Product].[Product Model Lines].[Model].
             &[Road-350-W] } ON 1
FROM (
  SELECT {
    ([SalesTerritory].[SalesTerritory].[Group].&[Europe])
  } ON 0
  FROM [Adventure Works]
)
WHERE { [Measures].[Internet Order Count] }
```

(a) Initial MDX repro.

```
WITH SET mySet1 AS DrilldownMember ((1)
    NameToSet ('[SalesTerritory].[SalesTerritory].[Group].
                 &[North America]')
  , NameToSet(1) ('[SalesTerritory].[SalesTerritory].[Group].
                  &[Europe]') )(1)
SELECT FROM ( SELECT(2) [Europe] ON 0(3)
              FROM [Adventure Works] )
```

(b) Final MDX min-repro.

Figure 1: Finding a min-repro for a complex MDX query.

$\mathcal{T}(r) = \checkmark$ means that the repro r passes the test (and therefore it does not reproduce the bug). The other two cases are used when some condition prevents getting a definite pass or fail result for a given repro. Specifically, $\mathcal{T}(R) = \circledS$ means that r is a syntactically valid repro but fails some semantic check (e.g., type checking), and $\mathcal{T}(r) = ?$ denotes any other unexpected condition. Some approaches in the literature (see Section 2) do not differentiate between $\circledS$ and ? values. As we explain in Section 3, this distinction enables a more focused search strategy.

Simplifications

At the core of any repro-minimization problem there is the notion of *simplifications*. Formally, a simplification is a function S : *repro* $\rightarrow$ *repro*, which transforms one repro into another that is "simpler". There are different variations of what constitutes a simpler repro. In this paper we consider the family of simplifications that return a subset of characters of the input repro[3]. Note that under this notion, the result of a simplification does not necessarily have to be syntactically correct. There are multiple ways of defining families of simplifications, and a contribution of this paper is a family based on language grammars (see Section 3 for details).

Min-repro Problem Statement

Consider a function $\mathcal{C}$ that measures the complexity of a repro (a natural definition of $\mathcal{C}$ is the length of the repro). Using the notation introduced earlier, we next define the repro minimization problem.

> Consider an initial repro r and a testing function $\mathcal{T}$ such that $\mathcal{T}(r) = \times$. Let $\mathcal{R}$ be the closure of r under simplifications. A *min-repro* for r is any $r^* \in \mathcal{R}$ such that (i) $\mathcal{T}(r^*) = \times$ and (ii) $\mathcal{C}(r^*)$ is minimal.

Since $\mathcal{T}$ is a black box with no exploitable properties, in the worst case the search problem is very difficult. In fact, there might be only one repro r^* in the closure $\mathcal{R}$ (besides the original one) that fails. We would then have to examine each repro in $\mathcal{R}$ to find r^* (which, for most simplification families, is exponential in the size of the original repro). We now discuss two approaches commonly used in practice to reduce the complexity of the problem [10].

The first approach is to assume monotonicity of test results. We specifically assume that if $\mathcal{T}(r) = \checkmark$, then $\mathcal{T}(S(r)) \neq \times$ for any simplification S. That is, if a repro r passes, then no simpler version of r would fail again. This is a very natural property of most real-world scenarios and allows specialized pruning strategies while searching for minimal repros.

Another common approach is to relax the notion of minimality into *1-minimality*. A repro r^* is 1-minimal with respect to a set of simplifications $\mathcal{S}$, if (i) $\mathcal{T}(r) = \times$ and (ii) $\mathcal{T}(S(r)) \neq \times$ for each $S \in \mathcal{S}$. This notion corresponds to a local minimum (with respect to a family of simplifications), where the repro fails ($\times$) but any single simplification does not fail anymore.

1.2 Additional Examples

In the example in Figure 1, the testing function $\mathcal{T}(r)$ is defined as follows. We first attempt to execute the given input repro r in the server. If r executes normally, $\mathcal{T}(r) = \checkmark$. If, otherwise, the server crashes while executing r, $\mathcal{T}(r) = \times$. The other two possible outputs of $\mathcal{T}$ have the usual meanings. If we cannot execute r due to a semantic error, $\mathcal{T}(r)$=$\circledS$. Any other unexpected condition results in $\mathcal{T}(r) = ?$. This specification can be used for problems that result in server crashes. We now briefly discuss additional scenarios:

Wrong results: Consider a new build of a database system that results in different results from those of a previous release for a given repro. In this case, the testing function attempts to execute the input repro in both systems, collects results, and returns $\checkmark$ (respectively, $\times$) if both result sets agree (respectively, disagree). The conditions for $\circledS$ and ? are the same as before.

Optimizer changes: Again consider two releases of a database system that return vastly different execution plans for a given query. The testing function tries to optimize the input repro r using both optimizers, and returns $\checkmark$ or $\times$ depending on whether the respective execution plans are the same or not. A weaker alternative that is very useful returns $\times$ only if the resulting plans differ in more than, say, 10%.

Specific engine features: Suppose we want the simplest repro that exercises a given optimization rule in the server, or uses a specific physical operator. In this situation, the testing function needs to be defined specifically for each scenario, by programmatically identifying when the given condition is satisfied.

The rest of the paper is structured as follows. In Section 2 we review related work. In Section 3 we illustrate the main drawbacks of previous approaches and introduce our technique to find min-repros leveraging language grammars. In Section 4 we discuss domain-specific extensions. Finally, in Section 5 we report an experimental evaluation of our approach using synthetic and real data.

[3]Other functions that can remove characters from a repro but also add new information are outside the scope of this work.

2. RELATED WORK

The work most closely related to ours (and our starting point) is delta-debugging [10, 11]. Delta-debugging is an algorithm used to minimize domain-independent failure inducing inputs in software. It has been used in several domains, including `C` code and web-based client user actions. Using our notation, delta-debugging considers the input as a sequence of characters, defines simplifications that remove consecutive subsequences (or their complements) from the input repro, and produces a 1-minimal configuration with respect to the family of simplifications considered. Delta-debugging an input repro r, $DD(r)$, is done by evaluating $DD'(r, 2)$:

$$DD'(r,m)=\begin{cases} DD'(r_i,2) & \text{if } \mathcal{T}(r_i)=\times \text{for some } i \\ DD'(\hat{r_i},\max(2,m-1)) & \text{if } \mathcal{T}(\hat{r_i})=\times \text{for some } i \\ DD'(r,\min(2\cdot m,|r|)) & \text{if } m<|r| \\ r & \text{otherwise (done)} \end{cases}$$

where $r = r_1+r_2+\ldots+r_m$, $|r_i| \approx |r|/m$, and $\hat{r_i}$ is obtained from r by removing r_i. In words, delta-debugging tries to remove each of the m consecutive portions of size $|r|/m$ from the input configuration (and also their complements), recursing as long as any of these simpler configurations fail. When no simplified configuration fails, it increases the granularity m and tries again, until all single characters have been tried with no success. One of the drawbacks of delta-debugging arises from its very generality. Since the main algorithm is domain independent, most of the simplifications result in syntactically invalid repros. Additionally, it does not take structural considerations into account. For instance, consider the SQL predicate `(a = 1 AND b = 2)`. Unless we precisely remove either "`a = 1 AND`" or "`AND b = 2`", the resulting configurations would not be syntactically valid. Delta-debugging would keep increasing granularity until $m = |r|$, but we can see that no single character would result in a valid predicate. In this situation, delta-debugging returns the whole predicate as a (valid) 1-minimal configuration. There are extensions to delta-debugging that consider tokens rather than characters as the most granular element in the input sequence (the above scenario, though, would not benefit from this extension).

To partially address the limitations of the original algorithm, an hierarchical extension of delta-debugging (called *HDD*) was recently introduced in [8]. The idea is to leverage parse trees and thus take structure into account. *HDD* iteratively applies delta-debugging to each level in a parse tree. In the k-th iteration, the input is seen as composed of text fragments, one per node in the parse tree at level k. The original delta-debugging algorithm is applied to this input, and the resulting nodes (and its descendants) survive the round, but the remaining nodes are omitted from consideration in subsequent iterations. *HDD* is specially useful when parse trees are wide and have a clean structure. For instance, in programming languages, higher levels in the parse tree correspond to modules, class definitions, methods, blocks, and so on. It makes sense to have a hierarchical version of delta-debugging that, say, minimizes the set of modules that produce a problem by considering modules as the atomic element in the input. When the smallest subset of modules is identified, *HDD* minimizes the class definitions, then the method definitions, and so on. In our specific domain, we observe that SQL and MDX do not fit very well the characteristics expected by *HDD*. Specifically, parse trees for these languages are usually deep, have nodes with few children, and are generally highly heterogeneous (e.g., there could be both definition of tables in the `FROM` clause and columns in the `GROUP BY` clause at the same level in a valid SQL parse tree). This makes individual delta-debugging iterations less effective. As we show experimentally, our techniques leverage parse trees more effectively due to a better characterization of the space of simplification functions, and result in both fewer test calls compared to *HDD* and also smaller repros overall.

In the specific context of SQL, reference [9] introduces RAGS, a system to stochastically generate large numbers of SQL statements for stress-testing. After a failure has been produced and identified, the test case has to be simplified because RAGS typically produces rather long query strings. The simplification of the test case has the same motivation as our work, but differs in an important aspect. Minimization is conducted by reversing production steps that were used to generate the original failing query and checking whether the failure still occurs. Therefore, the minimization procedure is tightly bound to the generation process, and can only minimize queries that were produced using RAGS. In contrast, our techniques can minimize any input repro.

Reference [4] presents a visual tool to help minimize database repros. The objective of such a tool is to provide automated support for many manually-intensive tasks performed during minimization, including simplification transformations, a high-level script language to automate sub-tasks and guide the search, record/replay functionality, and intuitive representations of results and the search space. The system in [4] implements delta-debugging as the main script for automatic minimization, and our techniques can be used to extend the capabilities of such a system.

3. FINDING MIN-REPROS

In this section we present our approach to minimize database repros based on language grammars. We will motivate our approach using a very simple example. Consider the following grammar, where lowercase tokens and symbols are terminals nodes:

$$\begin{aligned} &\text{L} \rightarrow \text{L} \vee \text{L} \mid \text{L} \wedge \text{L} \mid \text{(L)} \mid \text{C} \\ &\text{C} \rightarrow \text{id = number} \end{aligned}$$

Suppose that our expression evaluation engine cannot handle the same variable appearing multiple times in an expression. It would therefore fail with predicate "$(a = 1 \wedge b = 2) \vee (a = 3 \wedge c = 4)$" because a is used twice. The simplest repro for this problem is "$a = 1 \vee a = 3$". Suppose we use delta-debugging to minimize the original repro (which is composed of 19 tokens). We can see that, except for a single case when the granularity $m = 2$, all test cases are malformed (see Figure 2 for the initial test cases evaluated by delta-debugging). The only case that results in a valid predicate occurs when $m = 2$. In this scenario, the resulting test cases would be "$(a = 1 \wedge b = 2)\vee$" and "$(a = 3 \wedge c = 4)$" (if the implementation of delta-debugging partitions the original 19 tokens into se-

```
initial   ( a = 1 ∧ b = 2 ) ∨ ( a = 3 ∧ c = 4 )     →   ×
m = 2     ( a = 1 ∧ b = 2 ) ∨                       →   ?
                              ( a = 3 ∧ c = 4 )     →   ✓
m = 4     ( a = 1 ∧                                 →   ?
                    b = 2 ) ∨                       →   ?
                              ( a = 3               →   ?
                                      ∧ c = 4 )     →   ?
                    b = 2 ) ∨ ( a = 3 ∧ c = 4 )     →   ?
          ( a = 1 ∧           ( a = 3 ∧ c = 4 )     →   ?
          ( a = 1 ∧ b = 2 ) ∨         ∧ c = 4 )     →   ?
          ( a = 1 ∧ b = 2 ) ∨ ( a = 3               →   ?
m = 8     ( a                                       →   ?
              = 1 ∧                                 →   ?
                    b =                             →   ?
                        2 ) ∨                       →   ?
                           . . .
```

Figure 2: Delta-debugging produces many malformed inputs.

quences of 10 and 9 tokens, respectively), or "$(a = 1 \wedge b = 2)$" and "$\vee(a = 3 \wedge c = 4)$" (if the implementations chooses sequences of 9 and 10 tokens instead). Independently of the choice, only one test case is well-formed (the other has a misplaced $\vee$), and the test case passes (since it would contain a single mention of a). Therefore, delta-debugging returns the original input as the min-repro.

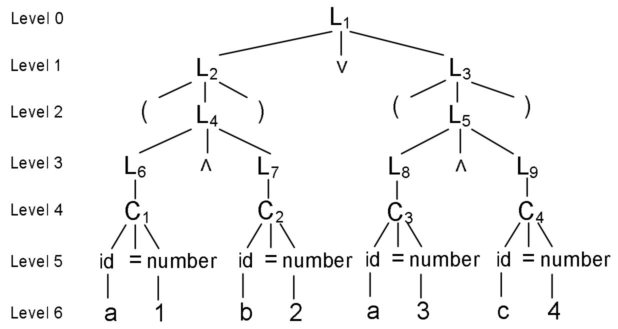

Figure 3: Minimizing repros with HDD.

Now consider hierarchical delta-debugging (HDD), which, up to a certain extent, takes structure into account. Figure 3 shows the parse tree of the original expression (note that the subscripts next to node labels are used for explanation purposes only, but are not actually part of the parse tree). HDD proceeds one level at a time, using traditional delta-debugging at each level. Level zero returns the root L_1, since the empty string does not fail. Level one cannot discard any of the three elements. Due to the operation of delta-debugging, level two also fails to remove parenthesis, since two simultaneous –not consecutive– removals are required to eliminate a pair of parenthesis. Thus, both sets of parenthesis survive level two and are included in the final answer. At level three, depending on the implementation of delta-debugging (specifically on how it segments the original six tokens in four approximately equally-sized subsequences) one of the superfluous L nodes (e.g., L_7 leading to $b = 2$ in the figure) would be removed. Note that after this node is removed, L_9 (leading to $c = 4$) would not be removed by delta-debugging. The net effect, after finishing level six, is that the min-repro might range from being the original expression (in the worst case) to being "$(a = 1) \vee (a = 3 \wedge c = 4)$" in case that the splits occur at the right tokens at level three.

Due to the specific family of simplifications that both the traditional delta-debugging technique and its hierarchical version consider, many interesting repros are not even explored. The simplifications considered by these algorithms are designed in this way because exploring every subset of tokens of the input repro is too costly. Thus, delta-debugging restricts the search space heuristically in such a way that is efficient to explore and at the same time finds some non-obvious repros.

Looking again at Figure 3 we realize that we can leverage the grammar that produced the input repro in a more systematic manner than what HDD does, by exploiting the knowledge of grammar rules themselves. If we replace one subtree rooted at L in level 1 in Figure 3 with another one rooted with the same label, we are guaranteed to produce a syntactically correct predicate. For instance, by replacing L_2 with L_6 and L_3 with L_8 in Figure 3, we obtain the desired min-repro. In the rest of this section we describe how to systematically explore this new family of simplifications and formally introduce our search strategy.

3.1 Grammar-based Simplification

We previously explained why existing techniques might not be effective to find min-repros due to the more or less arbitrary way in which they perform repro simplifications. We also hinted at the fact that the grammar encodes enough information to perform a much more focused set of simplifications. We next introduce a *local* approach to rewrite repros that works very well in practice and naturally leads to effective greedy variants.

Consider any internal node n in the repro's parse tree (e.g., let n be the root of the tree in Figure 3). Given n, we can identify all production rules in the grammar that originate from n's label. In our example, these would be $L \to L \vee L$, $L \to L \wedge L$, $L \to (L)$, and $L \to C$. Consider one of such rules, say $g = L \to L \wedge L$. Now, identify any three subtrees $\{n_1, n_2, n_3\}$ of n such that (i) the root of n_i is the same as the $i\text{-}th$ token in the right-hand-side of g, and (ii) the common ancestor of any subset of $\{n_i\}$ does not belong to any n_i (i.e., all n_i are disjoint). We can then replace n's children with n_i, obtaining a syntactically valid repro.

Formally, the simplifications that we consider for an input repro r are pairs (n, D), where n is a node in r's parse tree, and D=$[n_i]$ is a k-tuple of nodes in the subtree rooted at n (i.e., k descendants of n). Applying a simplification (n, D) to a parse tree is done by simply replacing all children of n by nodes in D (note that the original number of children in n and $|D|$ need not be the same). The set of simplifications for a repro r are as follows:

$$\mathcal{S}_r = \bigcup_{\substack{n \in \textit{parseTree}(r),\\ g \in \textit{grammarRules}(n)}} \mathcal{S}_n^g$$

where *parseTree*(r) corresponds to the parse tree of the input repro r, *grammarRules*(n) returns the set of production rules in the language grammar that have n's label in the left-hand side, and $\mathcal{S}_n^g$ are the simplifications for n and grammar rule g. For a node n with label T_n and grammar rule $g = T_n \to T_{n_1} T_{n_2} \ldots T_{n_k}$, we define:

$$\mathcal{S}_n^g = \{(n, D) : D \in \times_i \textit{candidates}(n, T_{n_i}) \wedge \textit{valid}(n, D)\}$$

In other words, we generate all combinations of subtrees among the *candidates* for each token in the grammar rule, and keep the *valid* ones. The set of *candidates* for a node n and token T_{n_i} is defined as the set of nodes $\{n_i\}$ in the subtree of n that have T_{n_i} as their labels. Since each candidate set is obtained independently from the others, there will be combinations that are invalid. Specifically, *valid*(n, D) discards combinations that either (i) exactly contain all children of n (that is, D=*children*(n)), or (ii) there exists d_1 and d_2 in D such that d_1 is a descendant of d_2.

EXAMPLE 1. *Consider an additional rule $L \to L \wedge C$ in the grammar of Figure 3. Then, candidates$(L_4, C) = \{C_1, C_2\}$ and candidates$(L_2, L) = \{L_4, L_6, L_7\}$. A simplification of the form $(L_4, \{L_6, \wedge, C_1\})$ is not valid because C_1 is a descendant of L_6.* ∎

To speed up the processing of results, we use numbering schemes originated in the context of XML query processing [2]. The idea is to associate each node n in the parse tree with a pair of numbers (l, r), which correspond to a pre-order (respectively post-order) traversal of the tree. Using this scheme, ancestor/descendant relationships (and thus the validity of a simplification as discussed above) can be checked in constant time, since n_1 is an ancestor of n_2 if and only if $l_1 < l_2$ and $r_1 > r_2$ (see [2] for more details).

```
minimizeGen (P:parse tree, in/out minP:parse tree)
01 for each simplification S=(n, D) of P // see Section 3.1
02   simpP = simplify(P, S) // replace n's children with D
03   if simpP is cached, continue (go back to 1)
04   $T_{simpP}$ = $\mathcal{T}$( treeToString(simpP) ) // cache result
05   if ($T_{simpP} \neq \checkmark$)
06      if ($T_{simpP}$=$\times$ and "simpP < minP")
07         minP = simpP
08      minimizeGen(simpP, minP)
```

Figure 4: Generic minimization strategy.

3.2 An Exhaustive Search Algorithm

We now introduce an algorithm to exhaustively traverse the search space of simplifications and obtain the globally optimal min-repro. Figure 4 shows a high level description of the minimization algorithm `minimizeGen`, which takes a parse tree representation of the original repro P as the input, and maintains and returns the min-repro `minP` (initially `minP=P` when calling `minimizeGen`, since by definition `P` fails). Conceptually, `minimizeGen` is a depth-first traversal of the set of repros obtained by using simplifications. The main algorithm iterates in lines 1-8 over each possible simplification `S` of `P` (see the previous section for a description of our family of simplifications). For each such simplification `S=(n,D)` line 2 obtains the simplified parse tree `simpP` by replacing children of `n` by `D` as explained earlier. If the resulting parse tree `simpP` has been seen before (note that multiple sequences of simplifications can result in the same parse tree) line 3 skips the simplification by using a global cache. Otherwise, `simpP` is processed to obtain the corresponding string, which is passed to the testing function in line 4 to obtain one of the possible answers $\{\checkmark, \times, Ⓢ, ?\}$. Due to monotonicity, every time that $T_{simpP} = \checkmark$ in line 4, no repro further simplified from `simpP` would fail, and thus we prune the search. If, instead, the testing function returns $\times$ and `simpP` is the smallest repro so far, it is saved in lines 6-7. In general, if the testing function does not pass (i.e., $T_{simpP} \neq \checkmark$), line 8 recursive calls `minimizeGen` with parse tree `simpP`. After all simplifications have been (recursively) processed, the algorithm returns the overall min-repro in `minP`.

Problem complexity

It can be shown that obtaining the globally optimal min-repro requires in the worst case evaluating a number of repros that is exponential in the original repro size. Consider the simple grammar:

$$\text{List} \rightarrow \text{Number} \mid \text{List} \text{ , } \text{Number}$$

which generates lists of numbers separated by commas. Consider a failing input repro with k numbers (which corresponds to a parse tree of $3k-1$ nodes). A simplified repro corresponding to any non-empty subset of these k numbers can be generated from the original one using a sequence of simplifications. Thus, there are 2^k-2 distinct repros (not counting the input one). Now consider any algorithm that attempts to minimize the input repro. Using an adversarial argument, we define the testing function in such a way that returns "?" for the first 2^k-3 distinct repros that the algorithm evaluates (whichever they are except for the original failing one), and $\times$ for the remaining one. Then, any algorithm has to evaluate 2^k-2 repros to get the correct answer for an input of size $3k-1$.

We now introduce two properties that significantly improve the performance of `minimizeGen` without compromising quality of results.

3.2.1 Redefining Simplification Candidates

Consider node L_1 in Figure 3 and rule $g = L \rightarrow L \vee L$. In this case, *candidates*(L_1, L)=$\{L_2, \ldots, L_8\}$. Suppose that we transform L_1 by replacing L_2 by L_6 and L_3 by any node in $\{L_3, L_5, L_7, L_8, L_9\}$. Although the simplification is valid, there is a different simplification path that results in the same repro. Specifically, we can first replace L_2 by L_4 and L_3 by the same element as above. Then, *candidates*(L_4, L) would include L_6 and we would obtain the same repro using an additional simplification. In general, generating the same repro multiple times results in performance degradation.

This situation arises because we are including in *candidates*(n, T) *every* node in the subtree of n with label T. Since we will apply simplifications in sequence, we can slightly refine the concept of candidates to eliminate *this specific* class of duplicate simplification paths (note that detecting *every possible* duplicate simplification path is in general non-computable). We then redefine the set of *candidates* for a node n and token T_{n_i} as the set of nodes $\{n_i\}$ in the subtree of n that (i) have T_{n_i} as their labels, and (ii) for which the only node (if any) in the path from n to n_i that has T_{n_i} as its label is a direct child of n. That is, the candidate set of n and T_{n_i} is the set of T_{n_i}-labeled nodes in the subtree of n that have no T_{n_i}-labeled ancestors other than n itself or its direct children. In this way, if a descendant n_i of n has an ancestor (other than a direct child of n) sharing the same token, n_i would be explored later, and thus we can remove it from the current candidate set in line 1.

3.2.2 Pruning Semantic Errors

In traditional and hierarchical delta-debugging algorithms (which do not differentiate Ⓢ and "?") most of the "?" outcomes result from syntactic errors (see Figure 2 for examples). A much smaller proportion of "?" outcomes result from semantic errors (i.e., cases in which the input string parses correctly, but there is a type error or some other post-parsing check fails). The (relatively insignificant) remaining fraction of "?" outcomes covers all other unexpected situations, such as for instance hitting other bugs, or problem-specific conditions for which we cannot evaluate a repro. Line 5 in Figure 4 correctly prunes all simplifications from a passing repro (i.e., T(`simpP`)=$\checkmark$) due to monotonicity. We next show that, under certain assumptions, we can also prune the search if T(`simpP`)=Ⓢ.

Note that by definition of our family of simplifications, all resulting repros are syntactically correct. Since we replace right-hand-sides of production rules with valid alternatives, the parser would always accept any simplified string. This property already eliminates the largest source of unknown outputs "?" in delta-debugging, thus focusing on repros that are actionable. That is not to say that *every* repro produced by our technique results in either $\times$ or $\checkmark$. Although parsing errors are ruled out, some repros result in semantic errors because our simplifications only take into account grammar rules.

EXAMPLE 2. *Consider the following SQL-based repro:*

```
SELECT * FROM R, S
WHERE R.x=S.y AND S.b=5
```

Suppose that the grammar rules for the list of tables in the `FROM` *clause are as follows (the actual SQL grammar is significantly more complex than what this example suggests, but we omit such details to simplify the presentation):*

$$\text{tableList} \rightarrow \text{tableList} \textbf{ , } \text{tableName} \mid \text{tableName}$$

Simplifying the top-most tableList *in the parse tree returns* `simpP`*:*

```
SELECT * FROM S
WHERE R.x=S.y AND S.b=5
```

Clearly, $\mathcal{T}$(`simpP`) = Ⓢ *due to the "dangling" column* `R.x`. ∎

Although errors from invalid inputs are still possible when using our technique, there is a crucial difference with the analogous case in delta-debugging. If the module that performs semantic checking satisfies the *reachable* property defined below, we can prune out repros that return Ⓢ without compromising the search strategy.

PROPERTY 1. *A semantic checker is* reachable *if, for any sequence of simplifications $R_1 \to R_2 \to \ldots \to R_n$ such that $\mathcal{T}(R_i) \in \{\checkmark, \times, Ⓢ\}$ there is another sequence of simplifications $R_1 \to R'_2 \to \ldots \to R'_k \to R_n$ such that $\mathcal{T}(R'_i) \in \{\checkmark, \times\}$.*

If a semantic checker is *reachable*, there is no need in further minimizing a repro R that returns Ⓢ, since there will be an alternative derivation path reaching anything useful that can be obtained from R. Therefore, we can replace line 5 in Figure 4 by:

```
05    if ($T_{simpP} \neq \checkmark$ and $T_{simpP} \neq$ Ⓢ)
```

As far as we can tell from inspecting the engine functional specification and source code, the semantic checkers in both languages that we experimented with (MDX and SQL) are *reachable*.

EXAMPLE 2. *(continued) The simplified repro* simpP *in the example can be further simplified into a semantically valid repro by eliminating the predicate $R.x = S.y$, thus resulting in* simpP2*:*

```
SELECT * FROM S
WHERE S.b=5
```

In this case, though, the same derivation could have been obtained by first eliminating the join predicate from the original repro:

```
SELECT * FROM R, S
WHERE S.b=5
```

and then removing table R*, obtaining the final repro* simpP2 *without passing through any* Ⓢ *values.* ▮

3.3 A Practical Search Algorithm

To obtain a practical algorithm, we first relax the global minimality condition to 1-minimality similar to the approaches in [8, 11]. Rather than finding the global min-repro, we would accept a locally minimum one (i.e., one for which every simplification does not fail). Note that even though both delta-debugging and our approach return 1-minimal solutions, the definition of 1-minimality always depends on a family of simplifications. For that reason, the min-repros obtained with our technique are simpler than those found using alternatives (see Section 5).

Figure 5 shows `minimize`, our algorithm to find 1-minimal repros. The idea is to adapt algorithm `minimizeGen` in Figure 4 to prune the search whenever $T(\texttt{simpP}) \neq \times$. Note that, since Ⓢ values can always be safely pruned without compromising quality due to the reachability property, the 1-minimality property only prunes, compared to `minimizeGen`, the considerable smaller fraction of repros that return "?". Algorithm `minimize` takes an additional parameter `LM`, which controls how greedy the resulting search strategy is. A value of `LM=1` is the default value that we use in the experimental evaluation and results in a purely greedy technique that explores a single local minimum. Increasing the value of `LM` results in `minimize` exploring additional local minima. Line 11 in Figure 5 checks whether we reached a local minimum by verifying whether some

```
minimize (P:parse tree // (note that $T_P = \times$),
         in/out LM:integer,
         in/out minP:parse tree)
01 isLM = true
02 for each simplification S=(N,D) of R // see Section 3.1
03   if (LM > 0)
04      simpP = simplify(P, S)
05      if (simpP is cached) continue back to 2
06      $T_{simpP}$ = $\mathcal{T}$( treeToString(simpP) ) // cache result
07      if ($T_{simpP} = \times$)
09         isLM = false
09         if ("simpP < minP") minP = simpP
10         minimize(simpP, LM, minP)
11 if (isLM) LM-
```

Figure 5: Greedy approach that ensures 1-minimality.

simplification from the current repro failed (i.e., whether `isLM` is true). In that case, the value of `LM` is decreased (line 3 stops the search once we processed the right number of local minima).

Complexity of `minimize`: Algorithm minimize either calls itself recursively at least once in lines 2-10, or otherwise reduces the value of `LM` by one. Consider the recursive call-tree of `minimize` for an input repro tree with N nodes. Any root-to-leaf path in the call-tree represents a sequence of simplifications starting from the original repro. Note that each simplification strictly reduces the number of nodes in the parse tree, so the height of the recursive call-tree cannot be more than N. Also, each leaf node in the call-tree corresponds to the case that no recursive call was made, and thus line 11 was executed decreasing the value of `LM`. Therefore, there cannot be more than `LM` leaf nodes in the call-tree. It follows that `minimize` is invoked at most $\texttt{LM} \cdot N$ times. Each such invocation considers, in the worst case, all possible simplifications of the input tree. Suppose that S is the maximum number of simplifications for a given node. Then, the maximum number of calls to the evaluation function in a single invocation of `minimize` is $N \cdot S$, and the total number of calls (counting recursive calls) is bounded by $\texttt{LM} \cdot S \cdot N^2$. The actual value of S depends on both the grammar rules and actual values in the tree. Although there are pathological scenarios for which S can grow large, in the context of SQL and MDX languages S is a small number. As we show experimentally, the bound $\texttt{LM} \cdot S \cdot N^2$ is very pessimistic, and our techniques use substantially fewer testing calls.

3.3.1 Ordering Simplifications

Due to the greedy approach of `minimize`, the ordering of simplifications can have a big impact in the quality of results. We use the following criteria to rank the possible simplifications of a given parse tree P in line 2:

Nodes: We first order all of P nodes using BFS, and generate simplifications in such an order. The rationale is that we start with simplifications that can make a big difference (i.e., nodes are higher in the parse tree) before going down into smaller and more precise ones. As an intuitive SQL example, we would first try to remove a whole sub-query. If this simplification does not reproduce the problem, we would go down the parse tree and try simplifying the sub-query itself. If, instead, removing the whole subquery still reproduces the problem, we had saved considerable time.

Grammar Rules: Within a given node n (obtained in BFS order), we use round robin on the grammar rules of n. Each time we are asked for a new simplification, we move to the next grammar rule that has outstanding simplifications and obtain the next one. The rationale is that in that way, we avoid getting stuck in a long sequence of simplifications of a "ineffective" grammar rule, and instead explore the space more evenly.

Candidates: For a given node and grammar rule, we sort candidates for each token in decreasing sub-tree size (i.e., by descending $(r - l)$ values , where l and r are given by the numbering scheme discussed in Section 3.1). We generate all combinations according to such order. This gives more priority to nodes with larger $(r - l)$ values, covering larger portions of the parse tree.

3.3.2 Identifying Breaking Changes

Looking back at Figure 1(b), we can see portions of text that, if removed from the min-repro, would prevent the bug from manifesting. We call these *breaking changes*, since they are fragments that, if added to a given passing test, would make the bug manifest. To obtain such breaking changes we proceed in two steps. First, we associate each parse tree P with the list of trees `simpP` that were simplified from P and resulted in $\mathcal{T}(\texttt{simpP}) = \checkmark$. Specifically, we add the following lines to algorithm `minimize` in Figure 5:

```
10.1    else if ($T_{\texttt{simpP}}$ = ✓)
10.2        P.simpPass += simpP
```

where `simpPass` is the list of passing repros associated with the current parse tree P. The result of this first step is that, after obtaining the min-repro, we already generated a (possibly long) list of all simplifications that result in a passing test. In the second step we post-process the resulting list of parse trees and return a *meaningful* subset of these. The following example clarifies this issue.

EXAMPLE 3. *Consider a generalized version of the motivating scenario of Section 3. Suppose that our initial SQL repro is:*

```
SELECT * FROM T
WHERE (a=1 AND b=2) OR (a=3 AND c=4)
```

and assume that the testing function fails whenever the same column is mentioned more than once in the query. The min-repro is:

```
SELECT * FROM T
WHERE a=1 OR a=3
```

The list of passing cases simplified from this min-repro contains:

```
SELECT * FROM T
SELECT * FROM T WHERE a=1
SELECT * FROM T WHERE a=3
```

Either of the last two statements (in conjunction with the min-repro) are useful in understanding the possible root cause of the problem. It seems, however, that the first one is "subsumed" by the others, since it is removing a strict superset of what the others do. ▮

The previous example contains just one unwanted repro, but for more complex scenarios there could be several of such superfluous alternatives. In general, we are interested in the set of changes that would make the problem disappear, and are not "dominated" by others. We then obtain the skyline [1] of the set of repros in `minP.simpPass`, for the following dominance function[4]:

$P_1 \prec P_2 \equiv$ *treeToString*(P_2) is subsequence of *treeToString*(P_1)

In the previous example, this definition successfully removes the unwanted entries.

[4]Dominance relationships over parse trees themselves are also possible. We chose our alternative since it is simpler and gives meaningful results.

3.3.3 Relaxing 1-minimality in `minimize`

Algorithm `minimize` in Figure 5 greedily finds 1-minimal repros by pruning those that do not fail (i.e., those that satisfy $T_{simpP} \neq \times$ in line 7). We can always safely prune repros that satisfy T_{simpP}=✓ due to monotonicity and those that satisfy T_{simpP}=Ⓢ due to reachability. However, `minimize` also prunes repros for which T_{simpP}=? in line 7. Therefore, even when `LP`=∞, `minimize` would not give the same results as the exhaustive `minimizeGen` of Figure 4. In this section we show how we can generalize `minimize` so that it behaves as the greedy variant when `LP=1` and as the exhaustive algorithm when `LP`=∞. For that purpose, we first associate each parse tree P with the list of trees `simpP` that were simplified from P and resulted in $\mathcal{T}(\texttt{simpP})$ =? (analogously to what we did in Section 3.3.2 for passing repros):

```
10.3    else if ($T_{\texttt{simpP}}$ = ?)
10.4        P.simpUnknown += simpP
```

We then recursively call the algorithm whenever the simplified repro satisfies $T_{simpP} \in \{\times, ?\}$. Specifically, we recurse with all the failing cases $(\times)$ first, and then with the unknown cases "?". The rationale is that chances are higher to reach a local minimum if we follow $\times$ first, as "?" cases are really unknown and subsequent simplifications could either pass or fail (note that while the ordering is irrelevant for an exhaustive enumeration, it makes sense for smaller values of `LM`). To implement this behavior, we simply add the following lines to `minimize`:

```
12  for each PSU in P.simpUnknown
13     if (maxSolutions > 0)
14        minimize(PSU, LM, minP)
```

These modifications expand the search space of `minimize` to consider additional simplifications from repros that are 1-minimal (and therefore not explored by `minimize`). While the complexity of the resulting algorithm is the same as that of `minimize` when `LP=1`, the worst case scenario even for `LP=2` is exponential using the adversarial argument of Section 3.2. In our experiments, we do not use this generalized version but instead rely on the original `minimize` algorithm in Figure 5.

3.3.4 Summary

To briefly summarize our techniques, the result of minimizing a repro R is a pair (R^*, B) such that:

1. $\mathcal{T}(R^*) = \times$ (R^* fails).
2. $\mathcal{T}(S(R^*)) \neq \times$ for any simplification S (R^* is 1-minimal).
3. B=$\{B_1, \ldots, B_n\}$ such that $\mathcal{T}(B_i) = \checkmark$, $B_i = S_i(R^*)$ for some simplification S_i, and $B_i \not\prec B_j$ for each B_i, B_j.

4. DOMAIN-SPECIFIC EXTENSIONS

In the previous section we introduced a minimization strategy that leverages the language grammar of the input repro. Exploiting this additional piece of information allows us to perform a more focused search and thus obtain better quality min-repros using fewer test calls than alternative techniques (see Section 5 for an experimental evaluation). Beyond the obvious requirement of such grammars, our technique is largely domain independent. For a given domain (e.g., SQL) there could be additional, domain-specific optimizations that further improve the performance of our techniques. In this section we give a high-level description of different ways to take advantage of domain-specific information. Details on these extensions are omitted due to space constraints and are the subject of future work.

4.1 Specialized Simplification Rules

There are scenarios that can benefit from either preventing certain simplifications to be applied, or conversely from applying simplifications not covered by our grammar-based approach. To illustrate the former case, consider the following MDX query:

```
SELECT [Date].[Calendar].[CalendarYear].[CY2001] ON 0
FROM [Adventure Works]
WHERE [Measures].[InternetStandardProductCost]
```

The MDX grammar allows us to further simplify this query by eliminating dimensions. A valid simplified query is shown below:

```
SELECT [CY2001] ON 0
FROM [Adventure Works]
WHERE [InternetStandardProductCost]
```

Due to certain features in the Analysis Services engine, both queries are actually equivalent. The reason is that the engine infers missing dimensions in hierarchies and therefore would implicitly reintroduce them while evaluating the query. However, developers trying to understand a problem would prefer the former query (with all the dimensions explicitly included) even though it is "larger" than the smallest possible repro. Otherwise, they would have to (manually) examine the cube and perform the inference of hierarchy dimensions. Handling this scenario can be accomplished by simply disabling production rules in the grammar that perform such simplifications (e.g., rules such as "*formula* $\rightarrow$ *formula . identifier*" should not be used for simplifications in line 2 of Figure 5). This general notion can be extended to be context sensitive, and thus a given production rule can have a complex condition that enables it to produce a simplification.

To illustrate the second scenario described above (i.e., using simplifications outside of our grammar-base approach), consider replacing the `SELECT` clause of a SQL query with the star symbol $*$. Because there are no $*$ symbols in the subtree of a `SELECT` clause that does not already use $*$, there will not be any valid simplification that produces such a change. But we know that in some scenarios this is valid (e.g., when there are no `GROUP BY` clauses in the query), so we could add such a simplification rule to the set of alternatives. This notion can be generalized to having default values for terminal nodes in the grammar (e.g., the value 0 for a number, or the string "foo" for a string identifier), and using such values in cases that there is no match for a given grammar rule. Using such extended rules can also help eliminate some Ⓢ results (e.g., if we understand the type system of the language, we can replace some subtree with a canonical constant value with the correct type).

4.2 Specialized Search

In Section 3.3.1 we detailed how we schedule the different simplifications for nodes, grammar rules, and bindings to candidate subtrees. We then explained the rationale behind our choices for a generic, domain-independent search. The more information we have about the repro domain (or even the specific bug for which we want a min-repro), the better we can bias the search towards better-quality repros. Consider, as an example, SQL as the underlying language, and suppose that the testing function actually executes the repro (i.e., it does not just optimize it). Removing predicates in SQL queries may result in very long running queries. In this case, it might be beneficial to rank first simplifications that result in *cheaper* execution plans.

A second search variant is related to the global search procedure. As explained in Section 3, the overall search follows a depth-first-search approach. That is, after exhausting all simplifications for a given node, we backtrack to its parent node and continue with the next simplification. Alternatively, we can perform a different strategy, by redefining the point to which we backtrack after exhausting a given node. Using destinations that are closer to the root generally increases the time to find a new local minimum, but at the same time results in more variety in the set of local minima.

Finally, an interesting variant results from knowledge of the grammar itself. Certain parser generators (e.g., ANTLR[5]) allow writing production rules using regular expressions. For instance, rather than writing production rules like:

```
groupByCols → column | column , groupByCols
```

we could use the more compact style (note that parenthesis and star symbols are meta-elements like "|" in the example above):

```
groupByCols → column ( , column )*
```

We can attempt a *best-effort approach* to detect these patterns in the grammar itself, and use them for specialized simplifications. As a simple example, consider the production rule above, which essentially specifies that a group-by clause contains a set of one or more columns separated by commas. Rather than generating all valid combinations as explained earlier, we could simplify `groupByCols` by (i) gathering all descendant *column* nodes in the subtree, (ii) use the traditional delta-debugging algorithm on this set of columns (fixing the appropriate set of commas for each case produced by delta-debugging). This specialized usage of delta-debugging would always return syntactically valid repros as it is driven by knowledge of the grammar rules. It can also be applied in other situations, such as for long scripts that contain multiple statements. For such cases, using traditional delta-debugging at the level of whole statements, and our technique within each statement can leverage the benefits of each approach at the right place.

4.3 Fixing Semantic Issues

We next describe a promising approach to significantly reduce the number of repros with semantic errors (i.e., those with $\mathcal{T}(R)$=Ⓢ). The general idea is to exploit domain-specific information about the semantic checker and directly "fix" a repro that would otherwise result in Ⓢ. As a simple example, consider a simplification rule that eliminates a table in the `FROM` clause of a given SQL query. Chances are that columns in the removed table are mentioned in the `SELECT`, `WHERE`, `GROUP BY`, `ORDER BY`, and `HAVING` clauses in the query, and therefore the simplified repro would be invalid. If we understand the semantics of SQL, we could fix the resulting repro by identifying the smallest subtree that (i) contains each mention of a column of the removed table, and (ii) can be removed without generating a syntactically incorrect repro. After such subtrees have been removed (the mechanisms for removing such subtrees are essentially the same ones as for applying simplifications), we check whether the removals resulted in another semantic problem, in which case we recursively fix the remaining problems.

EXAMPLE 4. *Consider as an example the repro below:*

```
SELECT R.a, SUM(S.b)
FROM R, S
WHERE R.x=S.y and S.b<10
GROUP BY R.a
```

and suppose that a simplification removes table `R`*. In this case, we first locate the mentions of columns in* `R` *in the query and eliminate the smallest subtrees that contain them. For* `R.a` *in the* `SELECT`

[5]See `http://www.antlr.org`.

clause we simply eliminate the column. For `R.x` *in the* `WHERE` *clause we eliminate the whole join predicate. Finally, for* `R.a` *in the* `GROUP BY` *clause, we need to remove, in addition to the column, the* `GROUP BY` *clause itself, because there cannot be a* `GROUP BY` *clause without columns. The result is the following syntactically valid repro:*

```
SELECT SUM(S.b)
FROM S
WHERE S.b<10
```

∎

5. EXPERIMENTAL EVALUATION

We now report an experimental evaluation of our techniques. We implemented all techniques in C# sharing as much code as possible. The same code base is used for different target languages (in our experiments, MDX and SQL running on Microsoft SQL Server 2008) by pointing at the corresponding language grammars. In our experiments we refer to our approach as *SIMP*. Specifically, we used the algorithm `minimize` of Figure 5 without the extensions of Section 3.3.3, and used a value of `LM`=1. We compare *SIMP* against the original delta-debugging algorithm [11] (denoted *DD* in the experiments), and the hierarchical variant of delta-debugging in [8] (denoted *HDD* in the experiments).

5.1 Comparison with Previous Approaches

In this section we contrast *SIMP* against the previously proposed delta-debugging variants. For that purpose, we use synthetic test cases (by synthetic we mean that we manufactured bugs as part of the testing function itself, even though the bugs do not appear in the actual database server). We next report in detail a small representative sample of a wider set of experiments we performed with different scenarios. We first evaluated the motivating example discussed in Section 3 and Example 3. For that purpose, we created a table T with three columns and an initial repro:

```
SELECT *
FROM T
WHERE (a=1 AND b=2) OR (a=3 AND c=4)
```

Since the database server does not fail for such query, we simulated the problem by implementing a testing function that, whenever the query can be optimized and executed, returns $\times$ whenever column `a` appears twice in the input query string. We then executed the different minimization techniques, which, as expected, returned the min-repros described in Section 3. Figure 6 summarizes the result of this simple experiment. Each column in the figure corresponds to one technique, and the bar measures the total number of test calls during the execution of the technique (for each technique we aggregate the counts grouped by the test result). Also, next to each technique name in the x-axis there is a reduction ratio, calculated as the number of tokens in the resulting min-repro divided by the number of tokens in the original repro[6]. On one hand, we can see that the reduction factor of *SIMP* is significantly higher (50%) than that of *HDD* (16%), which in turn is better than *DD* (which does not reduce the original repro whatsoever). At the same time, the total number of test calls for *SIMP* (17) is much smaller than those of *HDD* (168) and *DD* (85). The reason for both results is the focused family of simplifications in *SIMP* that results from leveraging language grammars. A significantly higher fraction of test cases in *SIMP* are either pass (✓) or fail (×) compared to other techniques.

[6]We used other measures of complexity, such as the number of nodes in the parse tree or the length of the repro string, and obtained very similar trends.

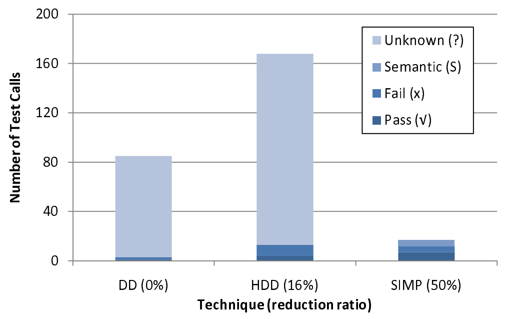

Figure 6: Minimizing a very simple query.

Since the previous query was very simple, we repeated the experiment with a real, complex TPC-H query[7]. Figure 7(a) shows the input TPC-H query #15. We used a testing function that fails whenever a (semantically valid) query contains more than one instance of column `l_shipdate`. Therefore, any min-repro must contain at least two mentions of the string `l_shipdate`. Figures 7(b-d) show, respectively, the min-repros obtained with *DD*, *HDD*, and *SIMP*. We can see that *DD* barely reduces the original query by only eliminating some syntactic sugar (e.g., the optional `AS` keyword for aliases) or some columns in the `SELECT` clause. *HDD* does a better job than *DD* but still retains a subquery, aggregate functions in the `SELECT` clause, superfluous function calls (e.g., `DATEADD`), and a `GROUP BY` clause. Finally, we can see that *SIMP* returns a truly min-repro, where all irrelevant query fragments are eliminated.

For the same experiment, Figure 8 reports the total number of test calls for the different minimization techniques. For this repro, *SIMP* requires an order of magnitude fewer test calls than the alternatives. (Note that in this case *HDD* performs better than *DD* since the original repro is complex enough for the hierarchical model of *HDD* to outperform the flat one of *DD*).

The last synthetically generated testing function that we report in this section uses the same initial repro in Figure 7(a). Suppose that the query engine has some deficiency in the way it handles nested sub-queries in `WHERE` clauses. We implemented such testing function by analyzing the parse tree of each semantically correct repro and failing if the nested subquery was present inside a `WHERE` clause. Figure 9 shows the resulting min-repros for different techniques. We see that not only *DD* results in a long min-repro (see Figure 9(a)), but the resulting repro is almost the same one as in Figure 7(b) for a different testing function! This shows that the traditional delta-debugging algorithm is not very appropriate for the domain of SQL queries. *HDD* fares better than *DD*, but still does not remove a superfluous aggregate function and an additional predicate in the `WHERE` clause, along with several redundant parenthetical expressions. *SIMP* again returns a significantly better min-repro than the alternatives, and it is not easy to find a smaller repro (even though there is a sub-query in the `FROM` clause, its result is needed in the nested `SELECT` clause). Figure 10 summarizes the number of test calls used by each technique. We see that *SIMP* results in the fewest calls overall (the numbers, however, are closer than for other experiments). As before, the fraction of actionable cases (i.e., those returning × or ✓) is significantly higher in *SIMP* than in *DD* or *HDD*. This is a good example for applying the extended techniques discussed in Section 4, which would remove many of the Ⓢ cases, resulting in a more efficient algorithm.

[7]Available at `http://www.tpc.org`.

```
SELECT s_suppkey, s_name, s_address, s_phone, tot_rev
FROM supplier,
     (SELECT l_suppkey AS supplier_no,
         SUM(l_extendedprice*(1-l_discount)) AS tot_rev
      FROM lineitem
      WHERE l_shipdate>='1997-03-01'
      AND l_shipdate<DATEADD(MM,3,'1997-03-01')
      GROUP BY l_suppkey) revenue
WHERE s_suppkey = supplier_no AND
      tot_rev = (
        SELECT MAX(tot_rev)
        FROM ( SELECT l_suppkey AS supplier_no,
                      SUM(l_extendedprice*l_discount)
                        AS tot_rev
               FROM lineitem
               WHERE l_shipdate>='1997-03-01' AND
                  l_shipdate<DATEADD(MM,3,'1997-03-01')
               GROUP BY l_suppkey ) revenue
             )
ORDER BY s_suppkey
```

(a) Original Repro.

```
SELECT s_suppkey
FROM supplier,
   ( SELECT l_suppkey supplier_no,
            SUM( l_extendedprice * (1)) tot_rev
     FROM lineitem
     WHERE l_shipdate>='1997-03-01' AND
           l_shipdate<DATEADD(MM,3,'1997-03-01')
     GROUP BY l_suppkey ) revenue
WHERE s_suppkey = supplier_no AND
      tot_rev = (
        SELECT (tot_rev)
        FROM ( SELECT SUM(l_discount) tot_rev
               FROM lineitem
               WHERE l_shipdate >= '1997-03-01' AND
                  l_shipdate<DATEADD(MM,3,'1997-03-01')
               GROUP BY l_suppkey ) revenue
              )
```

(b) Minimized repro using *DD*.

```
SELECT tot_rev
FROM ( SELECT SUM((l_discount)) tot_rev
      FROM lineitem
      WHERE l_shipdate >= '1997-03-01' AND
            l_shipdate < DATEADD(MM, 3, '1997-03-01')
      GROUP BY l_suppkey ) revenue
```

(c) Minimized repro using *HDD*.

```
SELECT l_suppkey
FROM lineitem
WHERE l_shipdate >= '1997-03-01' AND
      l_shipdate < '1997-03-01'
```

(d) Minimized repro using *SIMP*.

Figure 7: Duplicate column minimization for TPC-H query 15.

5.2 Case Studies

Having shown in the previous section that *SIMP* outperforms the alternative minimization approaches in both number of test calls and overall quality of the min-repro, we now focus on real-world scenarios that were handled using *SIMP*.

5.2.1 Analysis Services and MDX

We next comment on some scenarios in the context of Analysis Services and the MDX language that were handled using *SIMP*.

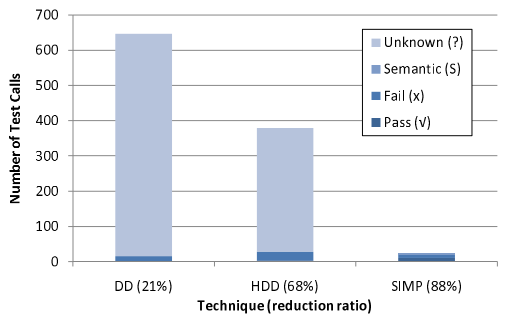

Figure 8: Duplicate column minimization for TPC-H 15.

```
SELECT tot_rev
FROM supplier,
   ( SELECT l_suppkey supplier_no,
            SUM( l_extendedprice * (l_discount)) tot_rev
     FROM lineitem
     WHERE l_shipdate>='1997-03-01' AND
           l_shipdate<DATEADD(MM,3,'1997-03-01')
     GROUP BY l_suppkey ) revenue
WHERE s_suppkey = supplier_no AND
      tot_rev = (
        SELECT (tot_rev)
        FROM ( SELECT SUM(l_discount) tot_rev
               FROM lineitem
               WHERE l_shipdate >= '1997-03-01' AND
                  l_shipdate<DATEADD(MM,3,'1997-03-01')
               GROUP BY l_suppkey ) revenue
              )
```

(a) Minimized repro using *DD*.

```
SELECT s_phone
FROM supplier,
     ( SELECT SUM ( ( l_discount ) ) tot_rev,
              l_suppkey supplier_no
       FROM lineitem
       GROUP BY l_suppkey ) revenue
WHERE s_suppkey = supplier_no AND
      tot_rev = ( SELECT ( tot_rev ) )
```

(b) Minimized repro using *HDD*.

```
SELECT tot_rev
FROM ( SELECT 1 tot_rev ) REVENUE
WHERE tot_rev = ( SELECT tot_rev )
```

(c) Minimized repro using *SIMP*.

Figure 9: Sub-query repro minimization for TPC-H query 15.

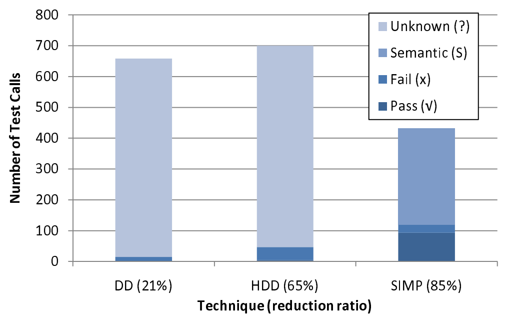

Figure 10: Subquery repro minimization for TPC-H 15.

Server Exceptions

In addition to the motivating example in Figure 1, *SIMP* was used to minimize a repro found during stress testing (using a query generator) that resulted in a stack overflow in the engine during query execution. After executing the repro, the server disappears and needs to be restarted (we coded such behavior as part of the testing function, which increased overall the time to minimize the repro). The original query spanned 69 lines, had over 6 levels of nesting in function calls, and overall was over 1.5Kb in length. A fragment of such query is shown in Figure 11(a). Every repro that resulted in the problem would require the engine to be restarted, and thus the time spent in the testing function was higher than in scenarios for which the server would recover itself after an exception. After around 20 minutes, *SIMP* produced the min-repro shown in Figure 11(b), isolating the problem as much as possible.

```
WITH SET mySet0 = BottomPercent (
  DrilldownLevel (
    IIf (
      IIf (
        IsAncestor([DimCurrency90].[RowNumber].&[104],
                   [DimCurrency90].[RowNumber].&[104])
        , IIf ( False, False, 0 )
      , TRUE
     )
    , Filter (
      Subset (
        { [DimCurrency90].[RowNumber].&[56] :
            :[DimCurrency90].[RowNumber].&[96] }, 6 ),
...   (57 additional lines)
```

(a) Original MDX query causing a server exception.

```
WITH SET mySet0 = [Latvian Lats]
SELECT FROM [Sandbox]
WHERE DrilldownLevelTop([DimCurrency90].[CurName], 3)
```

(b) Simplified MDX query causing a server exception.

Figure 11: Minimizing repros that result in server exceptions.

MDX Query Generator

Simp was also used (in a slightly different context) to debug the MDX query generator itself (which produced the query in the previous section). One requirement of such generator is to produce semantically correct queries (otherwise, most of the queries cannot even be parsed by the engine). It was observed that some complex queries produced by the generator resulted in semantic problems (for instance, in some corner cases there would be expressions with members belonging to different hierarchies, which is not allowed in MDX). To help debug the generator itself, we fed the failing queries to *SIMP*, and used a testing function that returned Ⓢ for all semantic errors *except* the one that the original query resulted in. We omit further details on these family of examples, but note that *SIMP* very quickly produced min-repros that were around 3 to 5 lines long, with an average reduction factor of above 75%.

5.3 SQL Server

We now describe some scenarios using *SIMP* in the context of the SQL language and Microsoft SQL Server.

Rule Testing

SQL Server's query optimizer is a top-down, transformational engine based on the Cascades framework [5]. At the core of the optimization framework there is a rule engine that transforms plan fragments into algebraically equivalent ones. Some rules are simple (such as join commutativity), and others are complex (such as view matching). In many situations, we want to analyze some transformation rule in detail, and thus need a query that both exercises the rule and for which the result of applying the rule is part of the final execution plan chosen by the optimizer. Of course, the smaller the query, the easier to isolate the rule application and understand its semantics. Figure 12(a) shows a query that benefits from a transformation rule in the optimizer (rule 323) that performs per-segment evaluation of relational expressions. Specifically, the rule introduces an operator that takes a relational input, segments it according to a set of columns, and applies a relational fragment to each successive segment in the relation. The precondition for applying such rule depends on specific properties of the query (e.g., there needs to be a common sub-expression that is joined with and without grouping columns). Additionally, the rule is effective if certain cardinality constraints are satisfied. We used *SIMP* with Figure 12(a) as a starting point and a testing function that fails whenever the optimization of the input query with and without enabling such transformation rule returns a different execution plan (in such case, we know both that the rule was applied and that the result is part of the final plan). Figure 12(b) shows the resulting min-repro, along with underlined fragments that, if removed, would make the rule ineffective. By inspecting the min-repro and the corresponding breaking changes, we can better understand the requirements of the rule. If needed, we can step into the rule code and understand its logic on a case that contains as few irrelevant fragments as possible.

```
SELECT SUM ( l_extendedprice ) / 7.0 AS avg_yearly
FROM lineitem, part, orders, customer, nation, region
WHERE l_orderkey = o_orderkey AND
      o_custkey = c_custkey AND
      c_nationkey = n_nationkey AND
      n_regionkey = r_regionkey AND
      l_partkey = p_partkey AND
      o_totalprice > 500 AND
      c_acctbal > 50 AND
      r_name = 'ASIA' AND
      p_brand = 'Brand#55' AND
      p_container = 'SM JAR' AND
      l_quantity < ( SELECT 0.2 * AVG(l_quantity)
                     FROM lineitem
                     WHERE l_partkey = p_partkey )
```

(a) Original query using rule 323.

```
SELECT l_extendedprice
FROM lineitem, part
WHERE l_partkey = p_partkey AND(1)
      p_brand = 'Brand#55' AND(2)
      l_quantity < ( SELECT AVG(3)(l_quantity)
                     FROM lineitem
                     WHERE l_partkey = p_partkey )(4)
```

(b) Simplified query using rule 323.

Figure 12: Minimizing repros to test optimizer rules.

N-Ary Join Stacking

SQL Server performs an initial heuristic join reordering of the input query to address situations in which the exhaustive enumeration engine times out. For that purpose, it first analyzes the query and identifies maximally connected components of join predicates (note that an input query can have semi-joins or group-by clauses, among others, that disconnect join-blocks). After components have been identified, the optimizer creates a special N-Ary join operator that contains all join relational inputs. Finally, different strategies reorder the children of N-Ary joins in different ways, to heuristically produce a good initial join order. Since this reordering does not work across N-Ary joins, it is important that each original N-Ary join covers maximally connected components in the query tree. To complicate the overall procedure, some query transformations that are applied concurrently with the construction of the N-Ary joins might eliminate certain "disconnecting" operators (e.g., group by clauses over inputs that are already distinct, or semi-joins that leverage primary/foreign keys). The net effect is that, for some corner scenarios, the optimizer logic fails to produce an N-Ary join over a maximally connected join graph. Instead, it results in an N-Ary join with another N-Ary join as a child. As a consequence, rather than heuristically reordering a single N-Ary join, the optimizer needs to perform two independent join reorderings over smaller sub-sets of joins, which decreases the quality of the resulting plans.

This bug was discovered as part of a code review, and from such review a repro was manually constructed (see Figure 13(a)). This repro operates over a schema with several foreign-key joins, and results in a stacked N-Ary join due to the removal of a semi-join. We used *SIMP* over such initial repro, where the testing function examines the intermediate result generated by the optimizer and fails whenever there is an N-Ary join on top of another N-Ary join. Figure 13(b) shows the result of the minimization process. We can see that, although the original repro was manually constructed and therefore thought of as already minimal, our techniques managed to further simplify it, eliminating redundant tables and join predicates. Note that any additional removal of tables or predicates ceases to reproduce the stacked N-Ary join problem.

```
SELECT *
FROM ( SELECT *
       FROM T2, T3, T4, T8
       WHERE EXISTS ( SELECT *
                      FROM T1
                      WHERE T2.fk21 = T1.pk1 AND
                            T3.fk31 = T1.pk1 AND
                            T4.fk41 = T1.pk1 AND
                            T8.fk81=T1.pk1 )
     ) T0, T5, T6, T7
WHERE T0.pk2 = T5.pk5 AND
      T0.pk3 = T6.pk6 AND
      T0.pk4 = T7.pk7
```

(a) Original stacked N-Ary join repro.

```
SELECT *
FROM ( SELECT *
       FROM T2, T3, T4, T8
       WHERE EXISTS ( SELECT *
                      FROM T1
                      WHERE fk21 = pk1 AND
                            fk81 = pk1 )
     ) T0, T6, T7
```

(b) Minimized stacked N-Ary join repro.

Figure 13: Minimizing a manually generated repro.

Configuration-parametric Query Optimization

As part of a project on automatic physical tuning, we built a component that extract the optimizer memo (i.e., the working space where all alternatives are evaluated) and enables very fast query optimization for varying physical designs (see [3] for additional details). The goal of this technique is to scale physical design tools that rely on repeatedly optimizing the same query under different physical configurations. Experimental evaluation showed that the technique in [3] can speedup query re-optimization by 30x to over 450x with virtually no loss in quality. To reach acceptable quality levels, we had to ensure that re-optimizations done by our tool were not far away from those obtained by the traditional optimizer. As part of the development of such a technique, we relied on *SIMP* to simplify queries for which our technique was inaccurate. We used a testing function that failed whenever the difference in cost between both approaches was over 5%, but omit details due to space constraints.

6. CONCLUSIONS

Database systems and database-centric applications are very complex systems. Reducing bugs and thus downtime is critical in such applications, but also a tedious process. Usually bugs are associated with repros, or deterministic sequences of steps that reproduce the problem. As repros can be very long and complex, the debugging process requires a preprocessing step where the initial repro is minimized. In this paper, we proposed a technique that automatically finds *min-repros* (i.e., the simplest version of a repro that still makes the original problem manifest). Our techniques are based on language grammars and therefore result in much more focused searches than previous approaches in the literature. By zooming into syntactically valid repros, we eliminate a large number of irrelevant repros that other approaches have to consider. We showed experimentally that our techniques are efficient, result in significantly smaller repros than alternative approaches, and have been successfully used in the context of two commercial query engines.

Acknowledgments

We thank Nicolae Mogoreanu and Nick Medveditskov from the Analysis Services team at Microsoft for valuable information about Analysis Services in general and MDX in particular, and for providing the MDX-centric examples used in this paper.

7. REFERENCES

[1] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[3] N. Bruno and R. Nehme. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2008.

[4] N. Bruno and R. Nehme. Finding min-repros in database software. In *International Workshop on Testing Database Systems (DBTest)*, 2009.

[5] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[6] Microsoft. Analysis services overview (white paper), 2007. http://download.microsoft.com/download/a/c/d/acd8e043-d69b-4f09-bc9e-4168b65aaa71/SSAS2008Overview.docx.

[7] Microsoft. MDX language reference (MDX), 2009. http://msdn.microsoft.com/en-us/library/ms145595.aspx.

[8] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering (ICSE)*, 2006.

[9] D. R. Slutz. Massive stochastic testing of SQL. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1998.

[10] A. Zeller. Yesterday, my program worked. Today, it does not. Why? *SIGSOFT Software Engineering Notes*, 24(6), 1999.

[11] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.