

FlexMin: A Flexible Tool for Automatic Bug Isolation in DBMS Software

Kristi Morton*
University of Washington
kmorton@cs.washington.edu

Nicolas Bruno
Microsoft Corp.
nbruno@microsoft.com

ABSTRACT

Debugging a database engine is an arduous task due to the complexity of the query workloads and systems. The first step in isolating a bug involves identifying a sequence of steps (or *repro*) that deterministically reproduces the problem. Repros are usually composed of various complex queries, which makes it very difficult to understand root causes and fix underlying bugs. Developers are thus burdened by the task of minimizing the repro as much as possible while still making it reproduce the original problem. In this paper we present FlexMin, a tool that automatically minimizes repros. FlexMin builds on previous repro minimization techniques, SIMP and delta-debugging, and makes two key contributions. First, FlexMin integrates previous approaches into a single, unified technique that performs simplifications in a more focused manner. This approach is flexible and can incorporate hints to guide the search. Second, FlexMin introduces data minimization as a new facet in the minimization process, which is helpful when data is too large or sensitive to share in its entirety. We used FlexMin to isolate bugs in a commercial DBMS, and show that it consistently produces simpler repros more rapidly than prior techniques.

Categories and Subject Descriptors

D.2.5 [Software]: Software Engineering — *Testing and Debugging, Debugging aids*

General Terms

Algorithms, Performance

1. INTRODUCTION

Database systems and applications are complex and debugging them is tedious and time-consuming. Even properly written queries can return unexpected results or behave in an unintended manner. The process of ensuring that software is bug-free involves rigorous cycles of testing and debugging. Typically the developer will run a series of tests

*Work done while the author was visiting Microsoft Research, Redmond.

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.

DBTest'11, 06-JUN-2011, Athens, Greece

Copyright 2011 ACM 978-1-4503-0655-3/11/06 ...\$10.00.

to uncover the presence of the bug and start the process of debugging once the problem is manifested for a test case.

In this work, we focus on complicated debugging scenarios where the query expressions in the applications are valid, but exhibit unexpected behaviors. These behaviors are difficult to identify or anticipate when testing the DBMS due to (i) test query expressions may use a variety of clauses which produce unexpected behavior only when used in very subtle or specific combinations, and (ii) data sets used by these expressions can vary substantially and may subtly influence the execution behavior of the query. These two factors influence the scope of a developer's effort to isolate a bug in the DBMS – the query clauses provide clues about which components of the software code base are involved, and the data sets influence the possible execution states of the running code. Our aim is to reduce the scope of possible code paths and execution states which trigger a bug by distilling the essential elements of the query and data sets involved.

We refer to any query expression that reproduces the bug caused by the original query as a *repro*. The original query is a repro for itself (trivially), but other query expressions can also reproduce the bug. In particular, since queries can be complex, a *minimum repro* is desirable. We define a minimum repro as the “simplest” version of the original query repro that still reproduces the bug. Furthermore, in some cases the data is key to reproducing the failure. However end users with extremely large or sensitive data may not be able to share the entire reproducing data set. Hence, such users would want to be able to isolate the failure-inducing data and query expressions necessary to reproduce the bug.

Recent work produced tools [1, 6, 9] that automate the process of isolating repros. There are two main research efforts in this area: a generic one (called delta-debugging), and a more focused one that relies on language grammars (called SIMP). These two approaches have complementary uses. Delta-debugging (DD) is effective at operating on a collection of homogeneous elements (e.g., independent statements in procedural code). Conversely, SIMP focuses on simplifying heterogeneous clauses within a single statement.

Our work unifies these two approaches based on the observation that database language grammars contain a mixture of heterogeneous and homogeneous structures. For example, a collection of independent SQL statements or a list of numbers may be delta-debugged in batch, while simplifying clauses with more complex structure (such as nested subexpressions) requires careful attention to grammar rules. We detect homogeneous structures by identifying recursive patterns in grammar rules. We also introduce a novel use of

DD for minimizing data by treating a data set as a collection of records with similar structure. Specifically, we make the following contributions:

1. Simultaneous minimization of input data and query complexity – all prior work has focused on the latter.
2. Better heuristics for guiding the minimization that avoids expensive and unhelpful local search.
3. Integrating user hints to guide simplifications.

Like prior work, FlexMin is able to handle any type of DBMS error whose result can be tested or checked for success or failure. For example, given an incorrect result returned from a new build of a database server, FlexMin executes the input repro on both the reliable server and the new build and compares the results. If the returned results disagree it attempts to reduce the input repro. However, bugs that exhibit nondeterministic behavior such as race conditions are beyond the scope of this paper. We evaluate FlexMin in the context of server crashes, queries that produce wrong or unexpected results, and execution plan changes across releases of a database system. FlexMin consistently found repros that were as small or smaller than the best ones produced by alternate approaches, and did so in less time.

2. PROBLEM STATEMENT

In this paper we assume that repros consist of an arbitrary number of query statements that form a workload. Each minimization scenario is associated with a *testing function* $\mathcal{T} : repro \rightarrow \{\checkmark, \times, \textcircled{S}, ?\}$, which determines whether the bug manifests for a given repro. Specifically, $\mathcal{T}(r) = \times$ means that the repro r fails the test (and therefore the bug is reproduced for r). Likewise, $\mathcal{T}(r) = \checkmark$ means that the repro r passes the test (and therefore it does not reproduce the bug). Furthermore, $\mathcal{T}(r) = \textcircled{S}$ means that r is a syntactically valid repro but fails some semantic check (e.g., type checking) [1], and $\mathcal{T}(r) = ?$ denotes any other unexpected condition (e.g., syntax errors).

Fundamentally, repro minimization involves applying simplifications that transform one repro in to another that is “simpler”. Formally, a simplification is defined as a function $S : repro \rightarrow repro$. As in prior work, we consider the family of simplifications that return a subset of characters of the input repro. Thus, the result of a simplification does not necessarily have to be syntactically correct.

2.1 Repro Minimization Problem Statement

Consider a function \mathcal{C} that measures the complexity of a repro (e.g., \mathcal{C} can be the length of the repro [1]). The repro minimization problem is then defined as follows:

Given 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 *minimal repro* for r is any $r^* \in \mathcal{R}$ such that: (i) $\mathcal{T}(r^*) = \times$ and (ii) $\mathcal{C}(r^*)$ is minimal.

In the worst case, the problem of finding the minimal repro is exponential in the size of the original repro [1]. However, there are two practical approaches which can reduce the complexity of the problem [9]. The first approach is to assume monotonicity of test results. That is, if $\mathcal{T}(r) = \checkmark$, then $\mathcal{T}(S(r)) \neq \times$ for any simplification S . In other words, 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. A second approach is to relax the notion of minimality into *1-minimality*. Formally, 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 $S \in \mathcal{S}$. In other words, 1-minimality corresponds to a local minimum (with respect to a family of simplifications), where the repro fails (\times) but any further simplifications do not fail.

3. MINIMIZING REPOS WITH FlexMin

In this section we present FlexMin, a grammar-based tool that automatically minimizes repros. Figure 1 shows the search mechanism in FlexMin that integrates both DD and SIMP into a single approach. At a high level, we take as inputs a parse tree representation of the original repro and a variable LM that controls the greediness of the algorithm. Additionally, we maintain and return the final minimum repro as `simplerTree`. Conceptually, we generate an exhaustive set of simplifications based on the grammar rules for each node, apply these simplifications to nodes in the parse tree, and prune the search whenever the test result doesn’t equal \times and skipping redundant simplifications through a caching mechanism. Like, SIMP, FlexMin processes the search space of simplifications based on grammar rules for each node to obtain a 1-minimal repro. However, FlexMin applies an improved enumeration strategy which dictates how the simplifications are ordered and applied. This approach prevents the algorithm from getting stuck on local minima. Additionally, since FlexMin integrates various simplification techniques, it is aware at a node level which specialized simplification techniques to apply and will try those in a depth-first manner. If any of these simplifications succeed in finding a smaller repro, FlexMin will continue to apply changes starting at the root of the specialized node. SIMP, on the other hand recurs at the root of the entire repro tree, which potentially results in a greater number of test calls. Figure 2 shows FlexMin’s dispatch mechanism that applies a specialized technique to the repro.

3.1 Improved simplification strategy

SIMP’s approach to enumerating and applying simplifications often wastes resources on dead ends. Suppose we have two `SELECT` statements in our initial repro. The first `SELECT` is more complex than the second, and the second `SELECT` is the one containing the bug. Due to SIMP’s enumeration strategy, it would first try all changes starting from the root node of the first `SELECT` before going to the next one. Hence, the algorithm may get stuck applying unnecessary transformations from a root node that will lead to local minima. To address this problem, FlexMin applies an improved, round robin-based approach to enumerating and applying the simplifications across all of the nodes in the tree. The basic idea is to generate a queue of ordered simplifications over all of the nodes in parse tree P . Our algorithm iterates over the P nodes using BFS, and for a given node n with grammar rule g , it queues the first x valid simplification candidates (where x is a parameter that can be specified by the user). This process continues over all the grammar rules for all of the nodes until there are no more candidates. Because this enumeration strategy applies simplifications in a more even fashion across all the nodes, it is less likely to get stuck on local minima. Figure 1 shows FlexMin’s primary simplifica-

```

minDriver (initialTree:parse tree, LM:integer,
            in/out simplerTree:parse tree)
01 ctx = new stack of Context
02 currentS = new queue of simplifications
03 currentS = enumerateRR(initialTree)
04 simplerTree, currentTree = initialTree
05 isLM = true
06 do
07   change = currentS.dequeue()
08   newTree = simplify(change, currentTree)
09   if (newTree is cached) continue to 24
10   testResult =  $T$ (newTree) // cache result
11   if (testResult =  $\times$ )
12     isLM = false
13     if (newTree < simplerTree) simplerTree = newTree
14     else if (newTree = simplerTree) continue to 24
15     ctx.push(new Context(currentTree,currentS,isLM))
16     currentTree = newTree
17     currentS = enumerateRR(newTree)
18     isLM = true
19     if (currentS.isEmpty() && !ctx.isEmpty())
20       prevCtx = ctx.Pop()
21       currentTree = prevCtx.currentTree
22       currentS = prevCtx.currentS
23       isLM = prevCtx.isLM
24 while (!currentS.isEmpty() && LM > 0)
25 if (isLM) LM--

```

Figure 1: FlexMin 1-minimal, greedy search.

```

simplify (change:simplification, currentTree:parse tree)
01 node = change.node
02 if (node.annotation = Ignore)
03   newTree = currentTree
04 else if ( node.annotation = DD)
05   newTree = deltaDebug(node, currentTree)
06 else if ( node.annotation = Data)
07   if(isLiteral(node))
08     newTree = deltaDebugLiteral(node, currentTree)
09   newTree = deltaDebugTable(node, currentTree)
10 else
11   newTree = simp(node, currentTree, change)
12 return newTree

```

Figure 2: FlexMin technique handler.

tion algorithm and includes the function `enumerateRR(P)`, which creates the queue of simplification candidates.

3.2 Integrating DD into FlexMin

FlexMin combines the DD and SIMP algorithms into a specialized, grammar-based approach. A key observation is that each of these two techniques, when applied to specific scenarios, can be more effective than their counterpart. For example, DD is effective in minimizing sequences of homogeneous elements (e.g., sequences of statements, or sequences of columns in a query). For such grammar rules, DD has, in the worst-case, $O(n^2)$ complexity [9], whereas SIMP can be $O(n^4)$, where n is the number of nodes in the sequence. FlexMin automatically detects from the grammar rules when it is appropriate to apply DD (and only applies it to recursive, repeated structures like lists and string literals).

DD partitions the input repro string r into m chunks, and attempts to reduce r by removing each of the m consecutive partitions of size $|r|/m$ from the input configuration (and also their complements). If any of these simpler configurations fail the algorithm recurs on them. However, if no simplified configuration fails, it attempts to reduce by trying a finer-grained partition m until all single characters have been tried. Furthermore, hierarchical delta-debugging [6], or HDD extends DD to minimize parse trees.

As a motivating example, consider an experiment from [1] in which the SIMP approach was able to find a smaller minimum repro than prior approaches, but did so at the cost of performance in terms of processing a large number of \textcircled{S} cases. We repeat the experiment using the same database server and TPC-H query 15 [8] (see Figure 3(a)). Since the database server does not fail for such a query, we simulated the bug as part of the testing function itself. In this example, the testing function produces the bug whenever it detects a nested subquery inside of a `WHERE` clause. Figure 3(b) shows the resulting repro found by the SIMP and FlexMin techniques. Note that this is the same repro that SIMP produced in the same experiment from [1]. In this case, SIMP made 708 total test calls with the following test result breakdown: $\times=18$, $\checkmark=127$, and $\textcircled{S}=563$. FlexMin resulted in 486 test calls: $\times=20$, $\checkmark=124$, and $\textcircled{S}=342$. FlexMin automatically applied the DD technique to the underlined parts of the TPC-H 15 query in Figure 3 and used SIMP on the rest. Since SIMP takes an aggressive approach in first trying simplifications that have the greatest impact on changing the tree (i.e. due to the BFS ordering), it can result in more semantic errors. In this case, DD prevents SIMP from being overly aggressive.

```

SELECT s_suppkey, s_name, s_address, s_phone, tot_rev(1)
FROM supplier,(2)
  (SELECT l_suppkey AS supplier_no,(2,3)
   SUM(l_extendedprice*(1-l_discount)) AS tot_rev(2,3)
  FROM lineitem(2)
   WHERE l_shipdate>='1997-03-01'(2)
   AND l_shipdate<DATEADD(MM,3,'1997-03-01')(2)
   GROUP BY l_suppkey) revenue(2)
WHERE s_suppkey = supplier_no AND
      tot_rev = (
        SELECT MAX(tot_rev)
        FROM ( SELECT l_suppkey AS supplier_no, (4)
              SUM(l_extendedprice*l_discount)(4)
              AS tot_rev(4)
            FROM lineitem
            WHERE l_shipdate>='1997-03-01' AND
                  l_shipdate<DATEADD(MM,3,'1997-03-01')(5)
            GROUP BY l_suppkey ) revenue )
ORDER BY s_suppkey

```

(a) Original Repro.

```

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

```

(b) Minimized repro using either *FlexMin* or *SIMP*.

Figure 3: TPC-H 15 subquery repro minimization.

3.2.1 Specialized adaptation of DD

Delta-debugging lists: The original DD algorithm produces many syntactically invalid repros when applied to languages like SQL because it does not have any structural information available to guide its minimization strategy. In FlexMin, we adapted the DD algorithm to work off the grammar, which we found greatly improves DD's efficacy in producing syntactically valid repros. The basic pattern of the grammar rule considered by DD is as follows:

$$L \rightarrow \alpha L \beta \mid \gamma$$

where α , β , and γ contain zero or more terminal or non-terminal nodes that do not produce L , a list node in the parse tree. When this pattern is detected at the top-most L in the parse tree, its corresponding node, `node`, is annotated as DD and `simplify(change,currentTree)` dispatches

`deltaDebug(node, currentTree)` on the subtree rooted at `node` in Figure 2. The function, `deltaDebug`, extends the DD algorithm in two primary ways. First, it only performs simplifications to `node` based on the grammar rules of the nodes in the subtree rooted at `node`. Additionally, after DD applies a simplification to `node`, it tests this newly simplified clause by inserting it back into the parse tree corresponding to the current minimum repro.

EXAMPLE 1. As a simple example, let us assume that we are given the following (already minimal) repro to minimize:

```
SELECT a,b,c,d,e,f,g,h FROM foo
```

Suppose that the grammar rules for the list of columns in the `SELECT` 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):

```
columnList → columnList , column | column
```

FlexMin has detected this grammar rule for the `columnList` node and applies DD starting with the top-most `columnList` in the parse tree. The specialized use of DD for repeated, homogeneous structures, generates syntactically valid repros and does so with fewer test calls than SIMP. In the above example, there were a total of 27 test calls, while the SIMP algorithm results in 43 total tests. SIMP has a worst case bound of $O(n^2S)$ where n is the number of nodes and S is the maximum number of simplifications for a given node. Then, for list structures such as the `columnList` in the previous example, containing n nodes with the `columnList` label and n nodes with the `column` label, SIMP would try (in the worst case) n^2 simplifications (S), resulting in an overall $O(n^4)$ complexity.

Delta-debugging data: Through our experimentation we also found out that DD was a natural fit for minimizing data, including string literals and tables. The divide-and-conquer approach is very effective in cases where the input repro contains repetitive sequences of items that are independent of each other. In other words, the removal of an item does not have any negative effect on the minimization. For example, the application of DD to a problematic string literal will systematically remove individual subsequences of characters until the smallest sequence of characters reproduces the failure. Because there is no structural context to consider in the simple case of removing characters from a string, the removal of subsequences of characters does not adversely affect the quality of the resulting minimization.

For minimizing relational data tables, DD proceeds at the granularity of removing individual rows or sequences of rows. We created an ordering over the rows by applying an artificial sequencing operation to the table. Our application of the DD algorithm creates temporary tables by partitioning the original table into m chunks. This is accomplished by issuing a `SELECT` of a subsequence of rows from the original table. Each new table will then get a temporary table id which will be incorporated back into the repro query expression for testing. If the test result is \times then DD proceeds to reduce this temporary table further, selecting out a smaller subset of rows and testing. However, if no simplified table leads to a repro, we increase the granularity of the partition and try to reduce until all single rows have been tried with no success.

3.3 User-directed search using language hints

While *FlexMin* automatically detects string literals from the grammar rules, it does not automatically detect candidate input data tables for minimization. *FlexMin* leaves the decision to the user regarding which data tables should be minimized and when this will take place relative to minimizing the repro query expression. Minimizing data tables can affect the time and outcome of query minimization, depending on how they are ordered relative to each other, so *FlexMin* relies on domain-specific information provided by the user through “hints”. We introduce a hinting language, which provides a mechanism for the user to supply such knowledge to guide the minimization. For example, the user may have an idea of where the problem might be, and thus convey additional insight through the hints, which help drive the minimization technique towards the minimum repro.

Prior approaches to repro minimization focused on minimizing along one dimension, which was the length of the query repro. When data is added to the mix, this approach is too naive, and query execution time is a dimension that also needs to be considered. Such domain-specific information can be conveyed through the hints to guide the order of the minimizations for data and the query.

The grammar for the hinting language is as follows:

```
topHint → Data,digit | DD,digit | SIMP,digit | Isolate,digit
topHintList → topHint topHintList | topHint
hint → Data,digit | DD,digit | SIMP,digit | Ignore
hintList → hint hintList | hint
```

The *topHint* refers to the toplevel technique that will be applied to the entire repro expression. There are four toplevel techniques: *Data* for minimizing tables, *DD* and *SIMP* for minimizing the expression, and *Isolate* for minimizing either a single expression or data table in isolation, using any of the aforementioned techniques. This type of hint is specified before the repro expression, contained within a block beginning with ‘/@’ and ending with ‘@/’. Additionally, the *hint* refers to hints that are nested inside the repro expression, and are contained within ‘/#’ and ‘#/’. Our approach allows the user to arbitrarily specify the simplification technique and priority, or order in which to apply that technique (i.e. specified as *digit* where 1 has the highest priority). Additionally, our hinting language supports nesting of hints. This functionality is achieved by first specifying a top level hint that will be applied to the entire expression, through the *topHint*. Then, the user can specify a technique override to a particular subexpression or table through the *Isolate* hint.

EXAMPLE 2. Suppose the user wants to minimize an entire repro expression using SIMP. However, she also wants to perform data minimization on one of the input tables before minimizing the expression since the input table, *foo*, is large. She would then annotate the repro expression as follows (underlined portions correspond to the hints):

```
/@Isolate,1 SIMP,2@/ SELECT a,b,c,d,e,f,g,h
FROM /#Data,1#/ foo
```

By using hints, the user can express different strategies to minimize the input repro expression, such as “*minimize the expression using SIMP/DD before/after minimizing the data.*”, “*Minimize one subexpression using SIMP/DD*”

before/after minimizing this other subexpression or data”, or “Minimize the entire expression using SIMP/DD except these specific subexpressions”.

4. EVALUATION

We implemented all techniques in C# and use the same code base for different target languages, specifically SQL and MDX. We compare FlexMin against SIMP, DD, and HDD in terms of how much the initial repro expression was reduced (reduction ratio), total time spent reducing (includes overhead of techniques), and number of test calls. As in [1], we compute the reduction ratio as the difference between the number of tokens in the initial repro and the resulting repro, divided by the number of tokens in the initial repro. In our evaluation, we demonstrate FlexMin and its distinct modes of operation. First, FlexMin refers to the fully-automatic query minimization approach described in Section 3. FlexMin-Data adds minimizing data tables through hints to FlexMin. Finally, the FlexMin-Hints mode allows the user to have complete control over the minimization strategy through the language hints from Section 3.3.

4.1 Minimizing Query Expressions

We first compare FlexMin against SIMP, HDD, and DD on minimizing SQL and MDX repro expressions. To that end, we use a mix of real, simulated, and synthetic repro minimization scenarios running on Microsoft SQL Server 2008. We define simulated to mean that the repro generated was for a real bug found for another database vendor, but because the bugs don’t manifest on our database system, we simulate their behavior in the testing function. Additionally, we define synthetic to mean that we manufactured bugs as part of the testing function, and the bugs were not previously known to manifest in a database system.

4.1.1 Simulated and Synthetic Bugs

TPC-H Query 15 Nested Subquery We first consider the motivating synthetic example from Section 3.2. In Table 1, FlexMin and SIMP have the same repro reduction ratio (85%), which is significantly better than that of HDD (65%) and DD (21%). Additionally, the number of test calls tells a slightly different story for SIMP, where it is much closer to HDD and DD due to a high number of semantic errors (563). The semantic errors are caused in part by the aggressiveness of SIMP’s enumeration strategy in which it first applies simplifications that will have the greatest impact on the overall size of the repro parse tree. In this example, the isolated effect of DD is evident through FlexMin, which resulted in 221 fewer semantic errors because it prevented the enumeration strategy from being too aggressive.

MDX Server Exception We simulate an actual bug from [1] that resulted in a server exception for MDX when run on Microsoft Analysis Services [4]. We created a testing function that simulates the behavior of the exception. In Table 1, we see that FlexMin and SIMP found the same minimum repro expression with an 81% reduction ratio, but FlexMin was able to do so with 107 fewer test calls. HDD fares better than DD, but was unable to remove some syntactic sugar from the last statement. Like the previous experiment, this test demonstrates the need for FlexMin, i.e. both a better enumeration strategy and specialized application of DD, to more quickly find a minimum repro expression. FlexMin found this minimum repro using a third of

the number of test calls of SIMP.

String Literal Minimization Consider a scenario in which the database engine fails on a query due to a problem with too many wildcard characters a string literal. In this experiment, we synthetically generate a testing function that fails on TPC-H query#16, which contains, ‘%Customer%Complaints%’, a string literal with multiple wildcard characters. FlexMin produced the following repro expression with the highest reduction ratio (87%): `SELECT s_suppkey FROM supplier WHERE s_comment LIKE ‘%’`. According to Table 1, Both HDD and DD were only able to reduce the initial repro by 51% and 46% respectively. SIMP produced a larger version of the minimum repro containing the full problematic string, ‘%Customer%Complaints%’. This example demonstrates that FlexMin can handle a wider variety of debugging scenarios than prior work.

4.1.2 SQL Server Optimization Testing

We evaluate the scalability of all techniques on large initial repro expressions. We consider a real-world scenario involving testing a query optimizer feature on Microsoft SQL Server 2008. The examples in this section (HpMerge64, HpMerge73, SplitMerge75, and RiMerge26) involve checking whether the same plan is produced by using (i) the traditional query optimizer, and (b) the query optimizer in a special mode. Specifically, we are evaluating a new optimizer feature that should not produce plan changes, and we start with a plan for which there is a plan change. The testing function tries to optimize the input repro using both settings, and returns ✓ or × depending on whether the respective execution plans are the same or not. The four input repro examples each contain between 43 to 50 lines of SQL and all have complex expressions; HpMerge73 contains a SELECT subquery nested seven levels deep.

Table 1 contains results for all four experiments. For these complex debugging scenarios, FlexMin consistently produced the smallest repro expression (reduction ratios from 80% to 89% for all four cases). SIMP was the next best, with reduction ratios ranging from 4% to 89%, and HDD and DD were unable to produce smaller repro expressions for all cases. FlexMin also took much less time to produce minimum repros (from 30 seconds to 6.7 minutes) than SIMP (9 minutes to over an hour) because it has a more focused search that results in significantly fewer test calls. FlexMin thus is a more scalable technique for minimizing complex query repro expressions. For all experiments, FlexMin has the best overall average for reduction ratio (84%) and minimization time (2.3 minutes).

4.2 Minimizing Data Tables and Expressions

In this section we study debugging scenarios where minimizing the query expression is not enough; rather the bug is related to a particular data item in the database (e.g., dirty data).

Divide by Zero Consider the following initial repro:

```
SELECT MAX(o_totalprice)/(CAST (MIN(o_totalprice) AS INT)/1000)
FROM orders
```

We implemented a testing function that returns × when the orders attribute, o_totalprice, is equal to -999.99 (a valid entry). The expression, (CAST (MIN(o_totalprice) AS INT)/1000), thus evaluates to zero, and the subsequent division operation is applied with a zero in the denominator.

Table 1: Query Minimization Experimental Results

Experiment	FlexMin			SIMP			HDD			DD		
	Reduction (%)	Time (sec)	Test calls	Reduction (%)	Time (sec)	Test calls	Reduction (%)	Time (sec)	Test calls	Reduction (%)	Time (sec)	Test calls
TPC-H 15	85	1	239	85	1	708	65	1	775	21	1	645
MDX exception	81	7	52	81	10	159	80	10	210	60	7	635
String literal	87	.05	30	83	.01	4	51	.83	514	46	.82	448
HpMerge64	84	404	37,998	4	3,601	92,680	0	1,254	28,139	0	250	55,825
HpMerge73	80	160	17,058	42	3,602	67,437	0	226	11,991	0	129	44,486
SplitMerge75	84	30	2,008	84	538	10,680	0	85	6,902	0	67	18,207
RiMerge26	89	357	6,572	89	505	38,182	0	879	14,656	0	12,713	31,834
Avg	84	137	9,137	67	1,180	29,979	28	351	9,027	18	1,881	21,726

```

/@Isolate,1 Isolate,2 SIMP,3@/
SELECT DISTINCT /#DD,1#/
a3.id, a3.fname, a3.lname, LOWER(a3.fname + ' ' + a3.lname)
actor_name, m0.name, m0.[rank], m0.[year],c0.role role0,
c1.role role1, c2.role role2, c3.role role3,
(SELECT COUNT(DISTINCT mid) FROM Casts WHERE role = c3.role)
cntd_movie3
FROM /#Data,2#/ Actor a0, /#Data,2#/ Actor a3, Movie m0,
Casts c0, Casts c1, Casts c2, Casts c3
WHERE
a0.fname = 'Kevin' AND a0.lname = 'Bacon' AND c0.pid = a0.id
AND c0.mid = c1.mid AND c1.pid = c2.pid AND c2.mid = c3.mid
AND c3.pid=a3.id AND NOT (a3.fname='Kevin' and a3.lname='Bacon')
AND NOT EXISTS (SELECT xc1.pid FROM Actor xa0,Casts xc0,Casts xc1
WHERE xa0.fname = 'Kevin' AND xa0.lname = 'Bacon'
AND xa0.id = xc0.pid AND xc0.mid = xc1.mid AND xc1.pid = a3.id)
AND m0.id = c3.mid AND m0.name LIKE 'Borat%'

```

(a) Initial SQL repro with hints underlined.

```

SELECT fname, lname FROM Actor_41

```

(b)SQL repro found only by FlexMin-Hints and FlexMin-Data.

Figure 4: SQL Turkish ‘I’ Problem.

In this case both the query and data table have valid entries, but the nature of the bug is more subtle. From Table 2, we see that HDD and DD are unable to reduce the original repro; and FlexMin-Data, FlexMin-Hints, and SIMP reduce it by 80% (returning `SELECT o_totalprice FROM orders` as the minimum repro). However, with SIMP the user has to manually examine 1.5 million rows of data to determine the offending value(s); FlexMin-Data and FlexMin-Hints do this automatically in a few seconds. The reported times for FlexMin-Data and FlexMin-Hints in Table 2 include minimizing the query and data, which is a total of 5-7 seconds. **Turkish ‘I’ Problem** Using IMDb data (www.imdb.com), we simulate the infamous Turkish ‘I’ Collation Problem. The Turkish language has 4 types of ‘I’ (*i.e.* lowercase and uppercase forms of both dotted and dotless ‘I’), which can cause difficulties in case folding, string comparison, and sorting. This is representative of a broader range of internationalization problems in DBMS software. Our synthetic test reproduces a failure for any query whose result set includes a Turkish ‘I’. Figure 4(a) shows the input repro tagged with user hints and 4(b) is the minimized repro found by FlexMin-Hints and FlexMin-Data. The user hints specify minimizing the `Actor` tables, based on an intuition that the query failure may originate with the international cast of characters from the movie *Borat*, which the query filters on. Both FlexMin-Data and FlexMin-Hints returned the smallest repro of all techniques (with a 97% reduction ratio) and did so with the fewest test calls (only 231 and 107 respectively), as shown in Table 2. FlexMin-Hints took 3.5 minutes and FlexMin-Data took 16 minutes to minimize the data and query. Minimizing the 1.9 million row `Actor` table after performing DD on the projection list helped FlexMin-Hints save time. FlexMin-Data minimizes the same two `Actor` tables as FlexMin-Hints, but specifies the data minimization to hap-

Table 2: Data and Query Minimization Results

Technique	Turkish ‘I’			Divide by Zero		
	Reduction (%)	Time (sec)	Test calls	Reduction (%)	Time (sec)	Test calls
FlexMin-Hints	97	213	107	80	7	32
FlexMin-Data	97	966	231	80	5	32
SIMP	72	946	269	80	0.2	10
HDD	54	907	4170	0	0.32	50
DD	12	110	1221	0	0.11	73

pen before the query minimization. This example shows the benefit of FlexMin-Hints, which allows data minimization to be interleaved with query minimization in multiple passes. Finally, SIMP reduces the initial repro expression by 72% (best of prior work), but produces a repro that references 6 of the 7 data tables. The user will thus have to manually inspect all 6 tables to find the offending records.

5. CONCLUSION

DBMSs and their applications are complex and debugging them is tedious and time-consuming. We presented FlexMin, a tool that automatically isolates bugs in DBMS software. FlexMin leverages language grammars and specially integrates two techniques from prior work, SIMP and DD, into a unified framework that can perform more focused simplifications. FlexMin can additionally minimize data, including string literals and input data tables. It also accepts guidance from the user regarding how to minimize a repro through a novel hinting language. We showed experimentally the merits of FlexMin in terms of being able to minimize a variety of repro expressions including data, and demonstrated that FlexMin more rapidly produced as good or better minimum repros than prior work.

6. REFERENCES

- N. Bruno. Minimizing Database Repros using Language Grammars. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2010.
- N. Bruno and R. Nehme. Finding min-repros in database software. In *International Workshop on Testing Database Systems (DBTest)*, 2009.
- N. Bruno and R. Nehme. Mini-Me: A Min-Repro System for Database Software. In *Proceedings of the 26th International Conference on Data Engineering*, 2010.
- Microsoft. Analysis services overview (white paper), 2007. <http://download.microsoft.com/download/a/c/d/acd8e043-d69b-4f09-bc9e-4168b65aaa71/SSAS20080verview.docx>.
- Microsoft. MDX language reference (MDX), 2009. <http://msdn.microsoft.com/en-us/library/ms145595.aspx>.
- G. Mishherghi and Z. Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering (ICSE)*, 2006.
- D. R. Slutz. Massive stochastic testing of SQL. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1998.
- TPC Transaction Processing Performance Council, 2010. <http://tpc.org>.
- A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.