Mini-Me: A Min-Repro System for Database Software

Nicolas Bruno #[1], Rimma Nehme *[2]

#*Microsoft Research*
Redmond, WA 98052 USA
[1]nicolasb@microsoft.com
**Microsoft Jim Gray Systems Lab*
Madison, WI 53703 USA
[2]rimman@microsoft.com

***Abstract*— Testing and debugging database software is often challenging and time consuming. A very arduous task for DB testers is finding a *min-repro* – the "simplest possible setup" that reproduces the original problem. Currently, a great deal of searching for min-repros is carried out manually using non-database-specific tools, which is both slow and error-prone. We propose to demonstrate a system, called *Mini-Me*[1], designed to ease and speed-up the task of finding min-repros in database-related products. *Mini-Me* employs several effective tools, including: the novel simplification transformations, the high-level language for creating search scripts and automation, the "record-and-replay" functionality, and the visualization of the search space and results. In addition to the standard *application mode*, the system can be interacted with in the *game mode*. The latter can provide an intrinsically motivating environment for developing successful search strategies by DB testers, which can be data-mined and recorded as patterns and used as recommendations for DB testers in the future. Potentially, a system like *Mini-Me* can save hours of time (for both customers and testers to isolate a problem), which could result in faster fixes and large cost savings to organizations.**

I. Introduction

Database software is complex along many dimensions, as it is comprised of a large number of features and execution components. An implicit assumption is that the underlying DBMS services are well-tested, reliable and correct.

Filling the Gap: Testing vs. Debugging

To ensure the absence of bugs in database software, testing and debugging are the two processes that are used hand in hand together. Testing can demonstrate the presence of a "bug," and debugging is used to identify what caused it and how to fix it. Too often, the starting point for the debugging process is a very large setup with a lot of irrelevant inputs and variables. This is a consequence of either randomized automatic test generators or real-world application scenarios. Of course, the shorter and more concise the setup to reproduce a problem is, the more likely the root cause of the problem is understood and is effectively fixed. Conceptually, we try to obtain a *min-repro* – the "simplest possible" version of the input variables that still reproduce the original problem.

[1]*Mini-Me*, a character in the *Austin Powers* movies, was the clone of *Dr. Evil* (the villain) and was identical to him in every way but was "one-eighth his size".

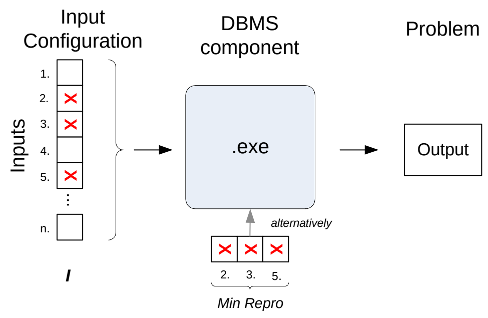


Fig. 1. Conceptual idea of a *min repro*.

Further removing or simplifying any input in a min-repro would stop the reproduceability of the problem.

Figure 1 illustrates the conceptual idea of a min-repro. Here an input configuration I on the left hand-side consists of a set of inputs $\{1...n\}$. A DBMS component shown in the middle takes this set of inputs and produces an output, considered by the DB tester a "problem". The configuration I may contain a lot of inputs that are irrelevant to the problem cause, i.e., their presence (or lack of presence) will not make any difference in whether the problem will appear or not. Hence, the DB tester needs to see only those inputs that are relevant to reproduce the problem (inputs 2, 3 and 5 in the figure). Furthermore, since inputs themselves can be complex (e.g., long SQL queries with nested sub-queries), it may be beneficial to find the simplest possible versions of the inputs both in the number and in their individual complexity.

Our Contributions

The contributions of our *Mini-Me* system are as follows:

1) **Repro Model**. We employ a general repro model that can capture different types of inputs, various database executables and a wide range of problem definitions.
2) **Transformations**. We design a set of novel simplification transformations in the database context that can incrementally simplify input configurations.
3) **High-Level Language**. We introduce a high-level language that can be used for creating customized min-repro scripts to re-use certain logic and to automate search sub-tasks.

4) **Record-and-Replay**. We present the “record-and-replay” functionality, where a sequence of actions can be recorded and then generalized into a min-repro search pattern that can be reused in the future (possibly in different contexts).
5) **Execution**. We describe how the search for a min-repro in *Mini-Me* can be performed in both the *application* and the *game* modes using the system’s intuitive UI.

II. Technical Details

A. System Overview

Figure 2 gives an overview of *Mini-Me* execution. First, a DB tester initializes the *Repro Function* (or short RF) – the abstraction that models the original repro comprised of a set of inputs, the database execution components[2] and the description of what the user views as the “problem” [1]. The min-repro system takes the RF object as its input (Step 1 in the figure), executes the min-repro search algorithm by interacting with the DBMS executable(s) (Step 2), prompts the DB tester for feedback (if applicable) to adapt its min-repro search strategy (Step 3), and at the end, returns a min-repro as a result (Step 4).

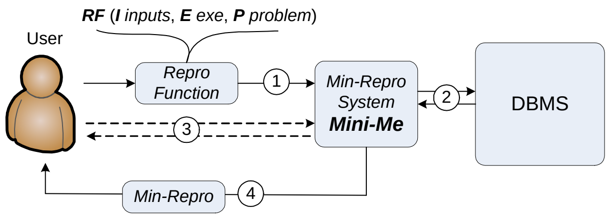


Fig. 2. Overview of *Mini-Me* execution.

In our work, we focus on two database-specific input types, namely, the *DML statements* (e.g., SQL queries) and the *physical structures* (e.g., indexes). However, extensions to support other input types can be added easily.

B. Modeling a Repro

The initial (large) repro and the problem description in *Mini-Me* is modeled using a *Repro Function* abstraction which is symbolically represented as $RF(I,E,P)$. RF provides a complete facility for users to specify their repro information and has the following three main parts: (1) a set of inputs I (e.g., a complex query workload and a set of available indexes), (2) a set of database executables E that consume I as inputs, and typically represent the database software where the problem originates, and (3) a set of rules P that describe the problem for which the min-repro is needed.

C. Simplifying Transformations

The minimization of a repro is carried out by executing *simplifying transformations* on the set of inputs I. We distinguish between two types of tranformations, namely the *inter-transformations* that are applicable to a set of inputs and the *intra-transformations* that are applicable to the “internal” content of an input.

[2]These are typically database executables.

1) Inter-Transformations: Table I shows the inter-transformations used in *Mini-Me* system. These transformations are defined as macros and can be applied to any input in a repro regardless of its type. The coarse-granularity of inter-transformations gives them an advantage of being able to quickly reduce the size of a repro in a few steps, which can be beneficial if the starting configuration contains many inputs. However, inter-transformations might not be as effective for repros with few but complex inputs [1] – the case where the intra-transformations are most helpful.

TABLE I
Inter-transformations

Input-Independent Inter-Transformations		
Macros	*Remove* *(Un)Lock* *Partition* (w/ *Choose*) - *manually* - *by n* - *randomly* - *by similarity* - *by rank*	1. Removes inputs 2. Makes inputs (im)mutable 3. Partitions inputs into subsets - defined by DB tester - into n subsets - randomly - by input similarity - by input rank

2) Intra-Transformations: Compared to the inter-transformations, the simplifying intra-transformations are more fine-grained and are input-specific. They are designed to operate on the internal contents of the repro inputs. For example, *query* inputs and *index* inputs have their own (tailored to their syntax and semantics) intra-tranformations as depicted in Table II.

TABLE II
Intra-transformations

Input-Specific Intra-Transformations	
Query intra-transformations	
Macros	1. SELECT simplification 2. FROM simplification 3. WHERE removal 4. WHERE simplification 5. GROUP BY simplification 6. GROUP BY removal 7. ORDER BY simplification 8. ORDER BY removal 9. Sub-query simplification 10. Sub-query removal
Custom	11. SQL parse-tree based
Index intra-transformations	
Macros	1. Column removal 2. Column order change 3. Column conversion (*key* ↔ *include*) 4. Column value change

D. Min-Repro Search Strategy

The general strategy for a min-repro search in *Mini-Me* can be described using the following key steps:

1) **Simplify**: This step executes a simplifying transformation(s) and returns a set of simpler input configurations (possibly containing only one set, if the partition inter-transformation was not used).
2) **Choose-to-Test**: In this step, some or possibly all of the simpler input configurations (from Step 1) are tested for problem reproduceability.
3) **Choose-to-Continue**: Among the simpler configurations that reproduce the problem (from Step 2), this step picks a configuration to continue with and goes to Step 1.

4) **Backtrack**: If nothing (from Step 2) reproduces the problem, the search backtracks to a prior input configuration, and then continues (with it) to Step 1.

E. High-Level Language

Mini-Me is equipped with a high-level language, called TLDB (short for *Test Language for Databases*), which allows users to create custom scripts and to re-use them in the future. TLDB uses XML as its primary syntax and is similar in spirit to the XEXPR [2] language with several extensions (functions and keywords) specific to database domain. Scripts encapsulate a general logic that can be then employed in the search for a min-repro in different scenarios. Existing debugging algorithms, e.g., *delta debugging* [3] and others can be easily implemented in TLDB.

F. Record-And-Replay Functionality

An intuitive way of debugging is when a user has tried a number of steps over time and they have reproduced the wanted results. *Mini-Me* features the "*record-and-replay*" functionality, which records user actions[3], generalizes them into a *pattern*, which is then available for *replay*, in either a manual min repro search or as a part of a script.

G. Visualization of Search Space and Results

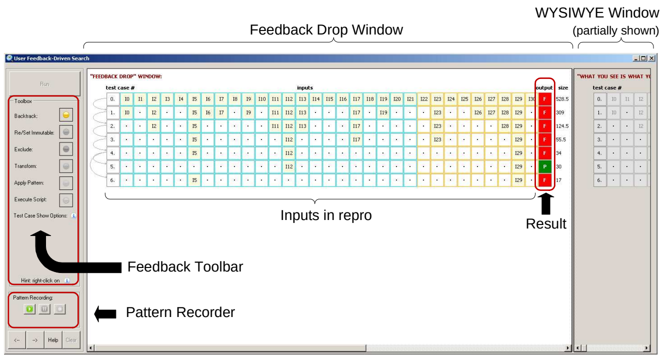
Simply knowing which repro is reproducing a problem is one thing, but presenting it in an intuitive and understandable manner (especially in complex scenarios) is another. *Mini-Me* provides a simple visualization of the search space and results, which can help a DB tester in understanding what and why might have caused a problem. Visualization can also facilitate in providing a better feedback by the users to the search strategy, thus creating a better "dialogue" between a tester and the min-repro search system, which can help find min-repros faster.

H. Execution Modes

1) Application Mode: Mini-Me running in the application mode executes as a normal application. Users specify a repro and then using the available in the system tools search for a min-repro. *Mini-Me* uses the concept of a *session* to distinguish among different attempts to find a min-repro. The session id can be used to reload a session (whenever needed) to continue the search. Sessions can also make the comparison of different runs/search strategies (for the same RF) possible.

2) Game Mode: An alternative mode of execution is the "game mode", where the search for a min-repro is presented in the form of a game. Games intrinsically motivate users to actively solve a problem. In [1], we've highlighted several essential elements of a "min-repro game" and how they are addressed in our system's game mode, e.g., customizable challenges, point system, etc. While in the game mode, *Mini-Me* can track and (in the background) data-mine for successful game strategies and generalize them into re-usable min-repro search patterns which can be used to provide recommendations, in the case a DB tester gets "stuck".

[3]The system also has a "pause" button, which allows to skip user actions if needed.



test case #	inputs																															output	size
0.	I0	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14	I15	I16	I17	I18	I19	I20	I21	I22	I23	I24	I25	I26	I27	I28	I29	I30	F	528.5
1.	I0	·	I2	·	·	I5	I6	I7	·	I9	·	I11	I12	I13	·	·	·	I17	·	I19	·	·	·	I23	·	·	I26	I27	I28	I29	·	F	309
2.	·	·	I2	·	·	I5	·	·	·	·	·	I11	I12	I13	·	·	·	I17	·	·	·	·	·	I23	·	·	·	·	I28	I29	·	F	124.5
3.	·	·	·	·	·	I5	·	·	·	·	·	·	I12	·	·	·	·	I17	·	·	·	·	·	I23	·	·	·	·	·	I29	·	F	55.5
4.	·	·	·	·	·	I5	·	·	·	·	·	·	I12	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	I29	·	F	34
5.	·	·	·	·	·	·	·	·	·	·	·	·	I12	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	I29	·	P	30
6.	·	·	·	·	·	I5	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	I29	·	F	17

test case #			
0.	I0	I1	I2
1.	I0	·	I2
2.	·	·	I2
3.	·	·	·
4.	·	·	·
5.	·	·	·
6.	·	·	·

Fig. 3. Search space visualization in *Mini-Me*.

III. Our Demonstration

A. Demonstration Scenarios

We will demonstrate *Mini-Me* using Microsoft SQL Server 2005 and 2008 and the following scenarios.

Scenario 1: Query Processor Cost Changes. This scenario depicts a situation when a DBMS vendor wants to determine a simple workload for which the cost estimates produced by the current and the previous versions of the query optimizer differ by more than a specified percent threshold δ (e.g., $\delta = 10\%$). This may indicate potential problems (variations) in the codebase of the newer optimizer[4].

Scenario 2: Physical Design Tuning Tools. The second scenario is in the context of physical design tuning. Given two different ways to perform *what-if* calls (specifically, the existing *what-if* API described in [4] and a new alternative described in [5]), we will show how a min-repro can be found for the case when the difference in the output is larger than a given tolerance, thus potentially indicating problems (or bugs) in the alternative approach.

B. User Interaction With Our System

The attendees of our demonstration will be able to interact with *Mini-Me* as follows:

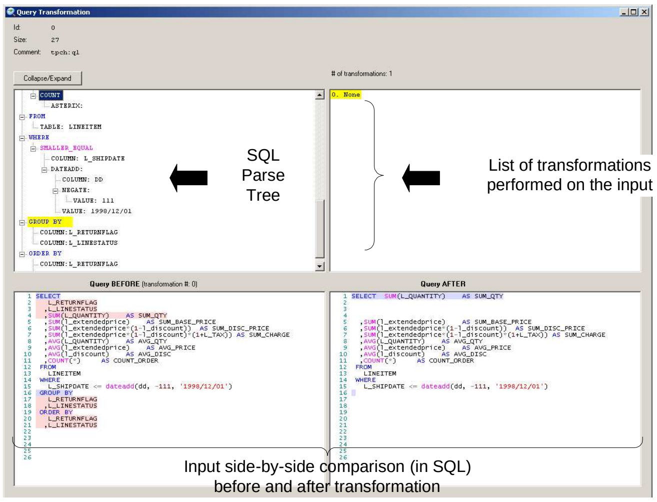
Specifying a repro: Users will be able to create a *Mini-Me* repro session through either a GUI form or an SQL statement. This will demonstrate how the system models a repro.

Manual search: Users will be able to search for a min-repro manually by providing their feedback to the system. Specifically, users will be able to perform the following actions: (1) *transform* – execute a simplification transformation on a repro, (2) *"record-and-replay"* functionality – record and then apply a search pattern, (3) *backtrack* – instruct the search to backtrack to a particular repro in the search space, (4) *execute script* – execute a script on a current repro.

Semi-automated search: To highlight the power for min-repro by running scripts created in our high-level language. We will also showcase the pattern "recording" example, that will be used for immediate "playback" in the search.

Visualization of search space results: Using the visualization tools in *Mini-Me*, we will graphically illustrate the search

[4]Similarly, it may be useful to determine queries for which the execution time differs by more than δ percent.


Query Transformation
Collapse/Expand
# of transformations: 1
SQL Parse Tree
List of transformations performed on the input
Input side-by-side comparison (in SQL) before and after transformation

Id: 0
Size: 27
Comment: tpch:q1

```
COUNT
  ASTERIX:
FROM
  TABLE: LINEITEM
WHERE
  SMALLER_EQUAL
    COLUMN: L_SHIPDATE
    DATEADD:
      COLUMN: DD
      NEGATE:
        VALUE: 111
      VALUE: 1998/12/01
GROUP BY
  COLUMN:L_RETURNFLAG
  COLUMN:L_LINESTATUS
ORDER BY
  COLUMN:L_RETURNFLAG
```

```
0. None
```

Query BEFORE (transformation #: 0)

```
SELECT
   L_RETURNFLAG
  ,L_LINESTATUS
  ,SUM(L_QUANTITY)     AS SUM_QTY
  ,SUM(l_extendedprice)     AS SUM_BASE_PRICE
  ,SUM(l_extendedprice*(1-l_discount))  AS SUM_DISC_PRICE
  ,SUM(l_extendedprice*(1-l_discount)*(1+L_TAX)) AS SUM_CHARGE
  ,AVG(L_QUANTITY)     AS AVG_QTY
  ,AVG(l_extendedprice)     AS AVG_PRICE
  ,AVG(l_discount)     AS AVG_DISC
  ,COUNT(*)      AS COUNT_ORDER
 FROM
   LINEITEM
 WHERE
   L_SHIPDATE <= dateadd(dd, -111, '1998/12/01')
 GROUP BY
   L_RETURNFLAG
  ,L_LINESTATUS
 ORDER BY
   L_RETURNFLAG
  ,L_LINESTATUS
```

Query AFTER

```
SELECT  SUM(L_QUANTITY)     AS SUM_QTY



  ,SUM(l_extendedprice)     AS SUM_BASE_PRICE
  ,SUM(l_extendedprice*(1-l_discount))  AS SUM_DISC_PRICE
  ,SUM(l_extendedprice*(1-l_discount)*(1+L_TAX)) AS SUM_CHARGE
  ,AVG(L_QUANTITY)     AS AVG_QTY
  ,AVG(l_extendedprice)     AS AVG_PRICE
  ,AVG(l_discount)     AS AVG_DISC
  ,COUNT(*)      AS COUNT_ORDER
 FROM
   LINEITEM
 WHERE
   L_SHIPDATE <= dateadd(dd, -111, '1998/12/01')
```

Fig. 4. Input transformations in *Mini-Me*.

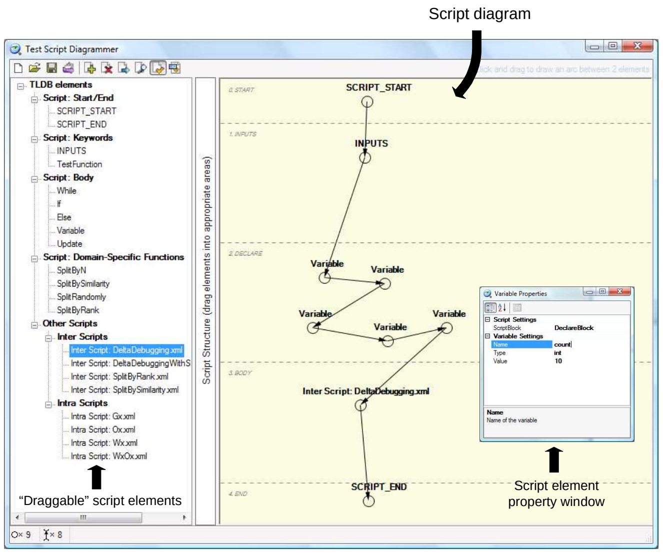


Fig. 5. Test script diagram in *Mini-Me*.

space, preview transformation changes, and show results of the search execution.

Execution in the game mode: Users will be able to interact with *Mini-Me* in the game mode (Figure 6). Using the points system we will describe how it increases users' motivation: for every transformation that successfully minimizes a repro a user will be given points. If a simplifying transformation caused the problem to be no longer reproduceable, some points will be deducted. A score summary following each "game session" also provides the users with their performance feedback, thus, facilitating progress assessment.

We will exploit the GUI tools for this demonstration[5].

Search space diagram: This window (Figure 3) illustrates the search space for a problem and has two uses: (1) the user feedback drop and (2) the "WYSIWYE" interface (*"WYSIWYE"* stands for "<u>W</u>hat <u>Y</u>ou <u>S</u>ee <u>I</u>s <u>W</u>hat <u>Y</u>ou <u>E</u>xecute". The former is used by users to "drop" their feedback to guide the search, and the latter previews the changes to the selected repro.

[5]Due to space constraints, we only list a subset of *Mini-Me* GUI tools here.

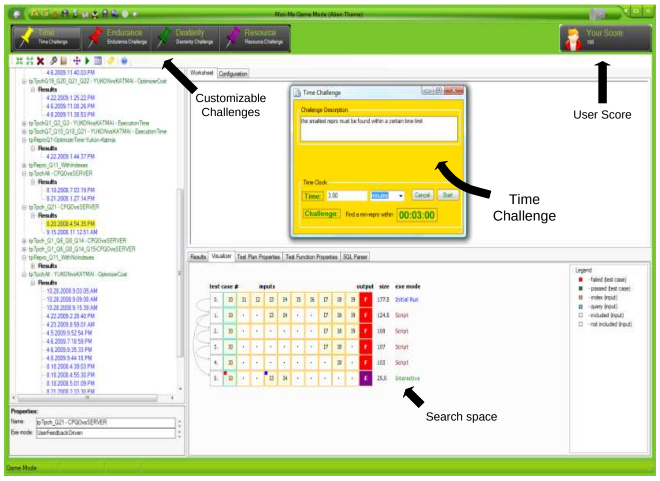


Fig. 6. *Mini-Me* in the game mode.

Input transformations: These windows enable the execution of transformations. Figure 4 shows an example of a query transformation and a side-by-side comparison of the query before and after the transformation.

Script diagram: This window (Figure 5) allows users to create diagrams of their test methods by simply dropping and connecting script elements to create an execution logic. Based on the diagram, the script code (in TLDB language) is generated. This interface allows users to create a sophisticated search logic very easily.

Execution component results viewer: This window (not shown) depicts the actual results returned by the execution component(s). Viewing these results can help users make a better judgement regarding how the search should proceed and what kind of feedback they should provide to the system.

IV. Conclusion

A great deal of current min-repro search is carried out manually using non-database-specific tools. An important aspect of reliable database services is the development of tools and techniques that can simplify the reproduceability of problems detected during testing to be fixed in debugging. Our min-repro system *Mini-Me* is precisely such a system, bridging the current gap between DB testing and debugging. *Mini-Me* is designed with both experienced and novice DB testers in mind. As a part of our system, we have suggested a game approach, inspired in part by the fact that humans enjoy "fun" applications. The game mode can enable less experienced DB testers to explore, learn and develop good min-repro repro search strategies.

References

[1] N. Bruno, R. Nehme, "Finding min-repros in database software," in *DBTest*, 2009.

[2] http://www.w3.org/TR/xexpr/.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[4] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database tuning advisor for Microsoft SQL Server 2005," in *VLDB*, 2004, pp. 1110–1121.

[5] N. Bruno and R. Nehme, "Configuration-parametric query optimization for physical design tuning," in *SIGMOD*, 2008, pp. 941–952.