Finding Min-Repros in Database Software

Nico Bruno
Microsoft Research
nbruno@microsoft.com

Rimma V. Nehme
Purdue University
rnehme@cs.purdue.edu

ABSTRACT

Testing and debugging database system applications is often challenging and time consuming. A database tester (or DB tester for short) has to detect a problem, determine why it happened, set up an environment to reproduce it, and then create a fix to resolve the problem. In many cases, problems appear in very complex scenarios, and thus the reproduction of a problem may be large and difficult to understand. This makes the task of finding the root cause of the problem very difficult. As a consequence, a very time-consuming task for DB testers is finding a *min-repro* – a process of weeding out irrelevant inputs and finding the simplest way to reproduce a problem. Currently, a great deal of searching for a min-repro is carried out manually, which is both slow and error-prone. In this paper, we present a system designed to ease finding min-repros in database-related products. The system employs a number of tools for min-repro search, including: novel simplification transformations, a high-level script language to automate sub-tasks and to guide the search, record-and-replay functionality, and an intuitive representation of results and the search space. These tools can save hours of time (for both customers and testers to isolate the problem), which could lead to faster fixes and large cost savings to organizations. Our min-repro system can be executed in two modes: (1) *application mode* and (2) *game mode*. The complexity and the tediosness of debugging has prompted us to explore the potential for a "game-like" approach to min-repro search. Inspired in part by the fact that humans enjoy "fun applications" and by the prevalence of long-term play of computer games, we believe that a game-like approach could help make the process of searching for a min repro more enjoyable and possibly help find min-repros faster.

1. INTRODUCTION

Database Systems Testing and Debugging

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.

To ensure bug-free data management services, 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 automatic randomized test generators, or real-world application scenarios. Of course, the shorter and more concise is the setup to reproduce a problem, 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 possible" version of the input variables that still reproduce the original problem. Further removing or simplifying any input in a min-repro would make the problem disappear.

Figure 1 illustrates a conceptual idea of a *min-repro*. Here an input configuration C on the left hand-side consists of a set of inputs $\{1...n\}$. A DBMS component illustrated in the middle takes this set of inputs and produces an output, considered (by a DBA or a DB tester) a "problem" or a "failure". The set of inputs in C 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 user needs to see only those inputs that are relevant to reproduce the problem (inputs 2, 3 and 5 in Figure 1). Furthermore, it may be far more beneficial to the user to see the "simplest possible" version of inputs in C in order to reproduce the same problem.

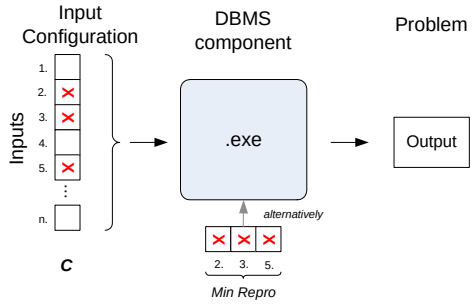


Figure 1: Min repro: main idea.

Consider the following concrete min-repro examples:
Example 1: *Query Processor Cost Changes.* A DBMS vendor may want 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 spec-

ified percent threshold δ (e.g., $\delta = 10\%$). Similarly, it may be useful to determine queries for which the execution time differs by more than δ percent.

Example 2: *Physical Design Tuning*. In the context of physical design tuning, given two different ways to perform *what-if* calls (specifically, the existing *what-if* API described in [1] and a new alternative described in [3]), a DB tester may want to find a min-repro, where the difference in the output is larger than a given tolerance thus indicating a problem in some of the alternatives, can be found.

Currently, there is a missing link between testing (where a problem is found but the repro can be very large and complex) and debugging (where the bugs are fixed). We propose to fill this gap in the database context with our proposed system in this paper.

Min-Repro Usage Scenarios

DBMS testing and debugging. In any testing or debugging domain, when it comes to problem repeatability it is essential to reduce a problem to the smallest and least complicated number of steps that can produce the bad result. Once the problem is reproduceable and the fix is created, the min-repro configuration may become a part of an automated test suite for future testing and verification that the problem is not recurring.

Benchmarking. Min-repros can be used to isolate the root cause of performance difference between successive releases of a database engine, or even to crisply contrast the performance/capabilities of different engines.

Privacy-preserving technical assistance. Often, enterprises encounter issues in their environments and need assistance from their database vendor. This naturally raises a number of legal and technical issues that must be addressed to preserve private and business-sensitive information through the control of the information flow amongst different entities. Min-repro can serve as a technical solution for preserving privacy in DBMS technical assistance. In order to not reveal business-sensitive information, an enterprise can create a smaller and simpler, and info-preserving configuration for the vendor to reproduce the same problem.

Challenges

- **Scope**. The scope of potential problems in database software is extremely large, making the task of designing a general system for finding min repros a challenging task.
- **Complexity**. Due to the complexity of the modern database management systems, potentially a significant number of parameters and their interaction may effect the search for a min-repro.
- **Non-determinism**. Similar problems don't guarantee to have the same min-repros.

Contributions

The contributions of our proposed system can be summarized as follows:

1. **Model**. We propose a general repro model that captures both inputs and a problem definition.
2. **Transformations**. We present novel simplification transformations.
3. **High-Level Language**. We present a high-level language for creating customized scripts for min-repro search.
4. **Execution**. We discuss how the search for a min-repro can be performed in both *application* and *game* modes.

2. MIN-REPRO SEARCH SYSTEM

2.1 Overview

Figure 2 illustrates the high level overview of the system execution. A DB tester creates a *user-defined test function* (or short UDTF), that models the original repro (e.g., a set of inputs, the execution components in the database and the specification of a problem). The min-repro system takes the UDTF as an input (Step 1 in the figure), executes the search algorithm interacting with the DBMS (Step 2), prompts the DB tester for feedback (if applicable) to guide the search (Step 3), and finally returns a min-repro for the problem specified in the UDTF as a result (Step 4).

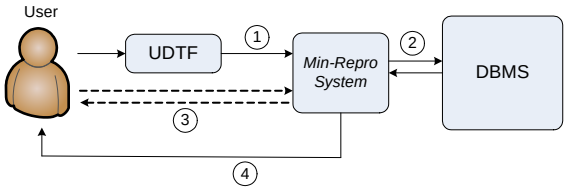
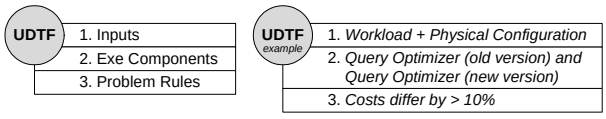


Figure 2: Overview of min-repro system execution.

For simplicity of presentation, we focus on two database-specific input types in repros, namely *DML statement* (e.g., SQL queries) and *physical structures*. For physical structures, we consider indexes, however other physical structures can be handled similarly. Indexes consist of a sequence of key columns optionally followed by a sequence of suffix (include) columns.

2.2 Modeling an Initial Repro and a Problem

The initial (large) repro and the problem is specified using a *user-defined test function* (UDTF). UDTF provides a complete facility for users to specify their repro information and has the following three main parts (see Figure 3 left): (1) a set of inputs (e.g., a complex query workload), (2) a set of execution components (e.g., successive releases of query optimizer in a database engine), and (3) a set of rules describing the problem (e.g., the new optimizer costs differ by more than 10% compared to the old one).



(a) Conceptual view (b) Example (query optimization)

Figure 3: Modeling a repro.

Users can specify UDTF in our system using XML language or using a declarative language like SQL.

2.3 Transformations

The modifications to a repro are carried out via *transformations*. We distinguish between two types of tranformations, namely the *inter-transformations* that are applicable to a set of inputs and the *intra-transformations* that applicable to the "internal" content of an input.

The inter-transformations supported by min repro system are illustrated in Table 1.

Removal: Any input i in a configuration C can be removed to obtain a new configuration $C' = C - \{i\}$.

Macros	1. Remove inputs 2. Make inputs immutable 3. Partition inputs

Table 1: *Inter* transformations

Immutability: Inputs i in a configuration C can be made immutable (to transformations). This may be useful, when no more simplification of certain inputs is desired.

Partitioning: Inputs i in a configuration C can be partitioned into a set of input groups to obtain a set of new configurations $C^* = \{\{C'\},\{C''\}...\}$ where $\forall\ C^{'} \in C^*$, $C^{'} \subset C$ and $\sum_{i=1}^{k} |C_i| = |C^*| = |C|$.

The intra-transformations are more fine-grained and input-specific, e.g., we have *query intra-tranformations* and *index intra-transformations*. The list of intra-transformations supported by our min-repro system is depicted in Table 2.

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 4. Column value change

Table 2: *Intra* transformations

In addition to transformations defined as macros, users can perform arbitrary intra-transformations on queries using *SQL parse tree* (see Figure 4). A visual representation of the query parse tree is exposed to the user, and the user can select a node in the parse tree and a transformation (e.g., edit, remove, simplify) to be applied to the node and its children.

2.4 Min-Repro Search Strategy

The general strategy for a min-repro search can be described as follows:

1. **Simplify**: *Input set is either partitioned into subsets or internal contents of inputs are modified. Both operations results in a "simpler" input configuration.*
2. **Test**: *The simpler configuration is tested. (In the case of partitioning, individual subsets are tested).*
3. **Choose**: *The search will continue with a simpler configuration (e.g., a subset) that reproduces the problem.*
4. **Backtrack**: *Otherwise, if the current "simpler" configuration does not reproduce the problem, the search should backtrack and try another simplification method.*

Users can control the search strategy, by manipulating the following logical steps: (1) *how to simplify* (e.g., how to partition input set into subsets and how to simplify each input), (2) *what to test* (e.g., which "simpler" subset to test), (3) *what to keep* (if multiple simpler configurations reproduce the problem, which configuration should the search continue with), (4) *when and where to backtrack* (if the problem can no longer be reproduced after a simplification, which earlier state to backtrack to).

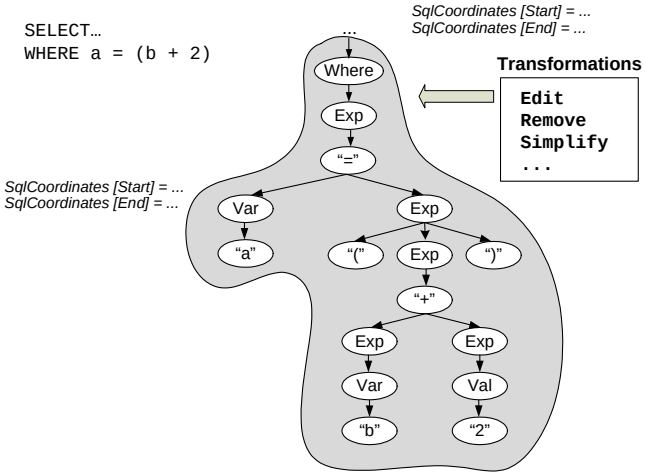


Figure 4: SQL Parse tree.

The strategies for simplifying, partitioning, testing and handling of multiple min-repros are depicted in Tables 3-6.

Strategies for **simplifying a repro** are shown in Table 3.

Heuristic	**Description**
Partition-First	partition configuration first, then test different subsets
Simplify-First	simplify individual inputs first and then procede with partitioning

Table 3: Simplification strategies

Table 4 illustrates **partitioning strategies** in a repro.

Heuristic	**Description**
Partition-by-n	partition into n subsets
Partition-randomly	partition randomly
Partition-by-similarity	partition by input similarity
Partition-by-rank	partition by a rank function

Table 4: Partitioning strategies

Below we describe the partitioning strategies in more detail.

Partition by n. This method breaks current input configuration into n subsets. If there are different input types, each type is partitioned into n subsets.

Partition randomly by n. Here, current input configuration is partitioned into n random subsets. One alternative is random partitioning, in which groups of inputs are formed by randomly selecting which input goes into which partition. The advantage here is that it is generally less work to construct test partitions.

Partition by similarity. Isolating problem-reproducing code changes can greatly profit from syntactic knowledge. All changes belonging to one class or one method can be combined, thereby reducing the amount of unresolved tests that occur during the minimization process. This is where a "similarity" function (per input type) becomes useful. This partitioning approach is similar in spirit to Equivalence Partitioning [6], Category Partition [5], and Domain Testing [2] which are based on the model that the input space of the test object may be divided into subsets based on the assumption that all points in the same subset result in

a similar behavior from the test object. This is called partition testing. Typically, in partition testing, the tester identifies test suites by selecting one or a few cases from each subset. The goal is to minimize the number of tests to run, yet to have a sufficient coverage.

Partition by rank. Inputs are characterized with respect to a certain *rank function* (e.g., input size), and then subsets are formed based on the rank of the inputs (e.g., all subsets must have a size $\leq \theta$, where θ is a size threshold).

Testing strategies (shown in Table 5) determine which subset(s) should be tested. This is where domain-specific combination strategies are useful. Search can benefit from choosing "interesting" (for the current problem specification) inputs combinations.

Heuristic	Description
Choose-random	random selection strategy
Choose-custom	custom heuristic

Table 5: Testing strategies

Table 6 depicts strategies **dealing with multiple independent inputs** that each reproduce the problem.

Heuristic	Description
First-Repro	Continue with the first failing subset
Smallest-Repro	Continue with the smallest failing subset

Table 6: Strategies to deal with multiple repros

2.5 High-Level Declarative Language

A high-level script language allows users to create custom scripts that are re-usable. The script language used in the min repro system, called TLDB (short for *Test Language for Databases*), uses XML as its primary syntax and is similar in spirit to the `XEXPR` language [4]. TLDB has several extensions (functions and keywords) specific to min repro problem domain. Using TLDB, test scripts can be created, and similar to transformations, can be applied to either a set of inputs (inter-scripts) or a particular input (intra-scripts). Scripts encapsulate a general logic that can be then employed in the search for a *min-repro* in different scenarios. Existing algorithms (e.g., *delta debugging* [8]) for instance can be easily implemented in TLDB and applies in the database contex.

Figure 5 illustrates a general structure of a custom script. Each script begins and ends with a tag `<Tldb>`. First, all inputs that are present in the current configuration[1] are specified and all variables used in the script are declared. All variables have a global scope and must be defined before the body of the script. The elements of the language are themselves XML tags, e.g., `<If>`, `<While>`, `<For>`, etc.

2.6 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. Our min repro system features "*record-and-replay*" functionality which records user actions, 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. Simplification patterns, similar to transformations, can be of two kinds: inter-patterns, as well as intra-patterns.

[1] *Current* configuration is the input configuration at the time of the script invocation. It does not need to be the initial input configuration with which the user has started the min repro search.

```
<Tldb>                          } TLDB start tag
<Inputs>...</Inputs>            } Inputs specification
<Variables>...</                } Variable declaration
Variables>
<Body>                          }
...                             }
<While>                         } Body of the script
...                             }
</While>                        }
</Body>                         } TLDB end tag
</Tldb>
```

Figure 5: TLDB script structure.

2.7 Search Space and Results Visualization

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. A singularly bad feature of many current debugging systems is the lack of attention that has been paid to the aspects of the debugging interface. A simple (yet intuitive) visualization of the search space and results can help DB testers in understanding what and why might have caused the problem. This can facilitate in users providing a better feedback to the search strategy, thus creating a better "dialogue" between a tester and the min-repro search system (especially in the game execution mode described in Section 3.2) and can help find the *min-repro* faster.

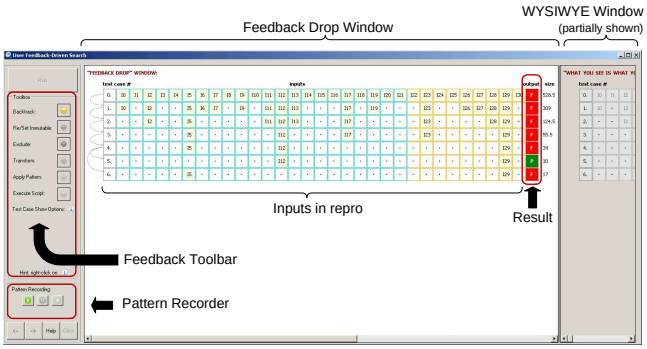


Figure 6: Search space visualization.

Figure 6 illustrates the window showing the search space for a problem. The search space diagram has two uses: (1) User feedback drop and (2) "WYSIWYE" interface (*"WYSIWYE"* stands for "What You See Is What You Execute" and is used to illustrate what will be executed, i.e., which changes would be performed to the selected repro). The feedback drop window is used by a user to "drop" the feedback to guide the search.

3. EXECUTION MODES

3.1 Application Mode

Application mode, as the name suggests, runs min-repro system as a normal application. Users specify a repro and using available to them system tools try to find a min-repro. Below we highlight several details regarding min-repro search in the application mode.

The system uses the concept of sessions to differentiate different attempts to find a min-repro. Each UDTF (a re-

pro instance) is executed in a separate session with a unique identity. The session information is saved and is associated with the UDTF. The session identifier can be used to reload the session as and when needed. Sessions make the comparison of different runs (for the same UDTF) possible. A session can be created through either a GUI form or an SQL statement.

Users can execute the search either manually or by integrating semi-automated tasks using scripts. The manual search for a min-repro uses feedback provided by the user as it becomes available. Specifically, a user can explicitly specify: (1) *transform* (to execute a simplification transformation on a repro), (2) *"record-and-replay" pattern* (to record and then apply a simplification pattern, which is a simple script containing a list of actions without any loops or conditionals predicates), (3) *backtrack* (to backtrack to a particular repro in the search space), (4) *execute script* (to execute a script on the current repro). System GUI facilitates in min-repro search, e.g., transformation window shows a side-by-side before and after comparison of an input for a particular transformation.

The power of min-repro system comes from performing a semi-automated search for *min-repro* by running scripts created in our high-level language. The script diagram form in the system allows users to create diagrams of their test methods by simply dropping and connecting script elements to create a logic. Based on the diagram, the script code (in TLDB language) will be generated. This interface allows users to create a sophisticated search logic very easily. The "pattern recording" supports an easy way of recording user actions that can be used for immediate "playback" in the search.

Using visualization tools in the system, the user can view the search space and the results of the search strategy. The execution component results viewer depicts the actual results returned by the execution component(s) (with their respective physical parameters). Viewing these results can help users make a better judgement regarding how the search should proceed, i.e., what kind of feedback they should provide to the system.

3.2 Game Mode

3.2.1 Why?

An alternative mode of execution is a "game mode", where the search for a min-repro is presented in the form of a game. Games naturally create an environment that intrinsically motivates users to actively solve a problem, while simultaneously providing entertainment and facilitating learning. There could be many motivations to a play game. Besides a DB tester's job responsibility (to find a min-repro), a DB tester may use the system game mode for exploration, proving oneself, mental exercise, and competition. The diagram below illustrates the potential for well-designed min-repro game:

$$\text{Game Mode} \rightarrow \text{Play} \rightarrow \text{Flow} \rightarrow \text{Motivation} \rightarrow \begin{array}{l}\text{Faster Learning}\\ \text{Enjoyment}\\ \textbf{\textit{Finding a Min Repro}}\\ \text{Competition}\\ \text{Bragging Rights}\end{array}$$

Games foster play, which produces a state of flow, which increases motivation, which ultimately leads to finding a min-repro in addition to learning, enjoyment, acknowledgement, etc. Furthermore, a game approach opens up possibilities for simultaneous learning in different database contexts. Players may learn from contextual information embedded in the dynamics of the game, the process generated by the game play, through the risks, benefits, costs, outcomes, and rewards of alternative strategies that result from decision making while playing a min-repro game.

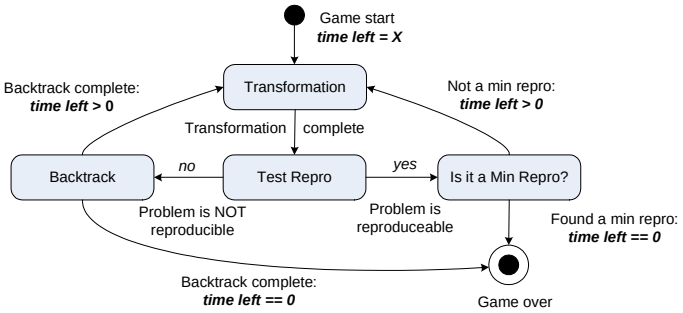


Figure 7: State diagram for min-repro game with time challenge.

3.2.2 Essential Elements of a Good Game

Making a good game is a challenging task. One of the core fundamentals of the games is that they are expected to be *fun*. Fun coould be defined as "the act of mastering a problem mentally," and could be born from the analysis of a problem and by solving that problem better and faster (e.g., through quantifiable outcome of getting a better time or a score or breaking a personal record).

Below we highlight the essential elements that are expected of a game, and how we address them in our system's game mode (depicted in bold).

1. Pursuing and achieving goals
   → ***Customizable Challenges***
2. Interactivity
   → ***User feedback drop, "WYSIWYE" interface***
3. Feedback about position relative to goals
   → ***Point System and Scoring Functions***
4. Interesting choices required to achieve goals
   → ***Simplification Transformations, Patterns, Scripts***
5. Consistency and fairness
   → ***Reliance on Database System Consistency***
6. Avoidance of repetition
   → ***Caching and Memoization***

Due to space limitations, we describe two key elements of a game mode in the rest of the paper, namely the customizable challenges and the point system.

3.2.3 Customizable Challenges in a Min Repro Game

The challenge is usually the central hub of a game play. The barriers that prevent the player from achieving that objective, are what determine the challenge. Below we highlight some of the challenges that can be enabled in a min-repro game mode:

- **Time Challenge**: The player is allowed only a certain amount of time to complete a task. A simple example is finding the smallest repro within a certain time limit. A state diagram for such a game is illustrated in Figure 7. This challenge can be combined with some other challenge.

- **Endurance Challenge**: This is the opposite of a timed challenge. Instead of having a limited amount of time to complete a task, an endurance challenge tests how far the player can go before he falters. In min-repro game, this challenge can test how far the user can go transforming inputs while still reproducing the problem. This can be very useful in practicing accurate weeding out of inputs that are likely not contribute to a problem.
- **Dexterity Challenge**: The player must accomplish some sort of feat that requires dexterity. It could be a mental challenge, where the player has to make quick decisions in order to overcome the obstacles he faces. In min-repro game, a dexterity challenge can be a choice for the most appropriate simplification transformation given a set of inputs in a repro.
- **Resource Control Challenge**: Many games use resource control as the challenge. The player is given a certain amount of a resource. He must use that resource to overcome an objective before it runs out. In min-repro game, a resource could be repro-specific, e.g., a bound on the memory size into which a repro should fit, or a max count of inputs desired in a repro.
- **Memory Challenge**: This type of challenge requires the player to know certain facts in order to win. In min-repro game, it can mean teaching the player some fact, like "transformation T_1 on average (from past runs) reduces repro size by 50%, while transformation T_2 only by 5%" and then making him recall that fact later on in the game, when given a choice between the transformations. Other examples include making the player memorize certain transformation patterns to execute the most efficient search strategy, or remember which types of intra-transformations work on certain types of inputs.

3.2.4 Point System

One of the most direct methods for motivating players is by assigning *points* for each instance of successful output (e.g., a smaller repro) produced during the game. For every transformation that successfully minimizes a repro a user is given points. If a transformation caused the problem to be no longer reproduceable, some points get deducted. Using points increases motivation by providing a clear connection among effort in the game, performance (achieving the winning condition), and outcomes (points). A score summary following each game also provides players with performance feedback, facilitating progress assessment on score-related goals (such as beating a previous game score and completing all challenges).

To enable the above, we establish a *point system* in the min-repro game for quantifying the merit of each possible user action (e.g., execution of a transformation or a test script or a recorded patern).

A general *scoring function* for a simplification transformation can be described as follows. The cost for a transformation in a repro should be based on complexity (or size) of transformation or the time it takes to execute it (otherwise, we might embark on exceptionally long routes to find a min-repro). The score of a simplifcation transformation is given by a formula of the following type:

$$S = -D+B, where \begin{cases} B = 0, if\ progblem\ is\ not\ reproduceable \\ B = D * 1 + \left(1 - \frac{|C'|}{|C|}\right) \end{cases}$$

where D is the cost of the transformation (determined by its size, or complexity or its time cost): we will refer to this as the base cost of a transformation; and B is the benefit of the transformation, which depends on $|C|$ – the size of the inputs configuration before the transformation and $|C'|$ – after the transformation. Correspondingly, $\left(1 - \frac{|C'|}{|C|}\right)$ represents the improvement in the size of a repro.

There could be several scoring mechanisms all working at the same time (e.g., one for inter-transformations and another for intra-transformations), and each being responsible for different strategic features of the game. Each separate scoring mechanism is then combined into one overall score. This can be as simple as adding scores together with a fixed weight for each. In this sense, scoring functions are like the tactical analysis: primitive tactics are combined into a more sophisticated view of the quality of the situation.

4. CONCLUSION

Testing and debugging are one of the most expensive and time consuming activities in any software development cycle, including database systems. The process of identifying and correcting a problem's root cause remains very labor-intensive, painful and costly to organizations. A DBA or DB tester has to detect the problem, determine why it happens, set up an environment to reproduce it, and then create a fix which must be confirmed to resolve the problem. A great deal of current testing is still carried out manually using general non-database-specific tools. An important aspect of reliable database services is the development of tools and techniques that can simplify problems detected in testing to be fixed in debugging. Our min-repro system is precisely such a tool, addressing the current gap between DB testing and debugging. As a part of our system, we have also suggested a game approach, inspired in part by the fact that humans enjoy "fun" applications and by the successful encouragement of long-term play of computer games. Although there are several examples of successful "games with a purpose" [7], none of them focus on database software, and in particular on min-repro problem.

5. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*, pages 1110–1121, 2004.

[2] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[3] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 941–952, New York, NY, USA, 2008. ACM.

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

[5] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[6] S. Reid. The art of software testing, second edition. *Softw. Test., Verif. Reliab.*, 15(2):136–137, 2005.

[7] L. von Ahn. Games with a purpose. *Computer*, 39(6):92–94, 2006.

[8] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.