Shaman: A Self-Healing Database System

Songyun Duan, Peter Franklin, Vamsidhar Thummala, Dongdong Zhao, and Shivnath Babu
Department of Computer Science, Duke University, USA
{syduan,pjf5,vamsi,zhao,shivnath}@cs.duke.edu

1. Introduction

A self-healing system is a grand-challenge vision where the system will detect, diagnose, and repair performance problems and hardware/software faults automatically [3]. These systems take humans out of the failure-recovery loop, enabling recovery to happen at fast machine timescales rather than slower human timescales. Are self-healing database systems utopia or just a hard puzzle to solve? We argue for the latter and propose to demonstrate a self-healing database system, called Shaman, that is being prototyped at Duke.

An important trend over the last few years has been the emergence of mechanisms that act as enablers for self-healing. (Policies for self-healing are an entirely different story which will be discussed momentarily.) Many of these mechanisms have emerged because of the decreasing tolerance towards service downtime.1 We begin by pointing out some of these mechanisms:

- Database systems have been making more and more of their configuration parameters reconfigurable dynamically. For example, until a few years back, changing the buffer pool size in many database systems required a database restart; but not any more.
- Many database systems can now dynamically assimilate new hardware resources without any application downtime. For example, “Hot-add CPU” feature in Microsoft SQL Server 2008 enables plugging in a new CPU while the database is running; the database will be reconfigured online to make use of the new CPU. “Hot-add memory” was introduced in SQL Server 2005.
- Many database systems can now run administrative tasks online in a throttled mode without any unbounded effect on the production workload. With this utility, resource-intensive tasks like index creation or rebuild and table/index fragmentation can be done online.
- Perhaps the biggest enabler for self-healing is the rapid rise and adoption of Virtual machine (VM) technology. The leading VM systems (e.g., VMware, Xen) support live migration, checkpoint/restart, and fine-grained allocation of server resources as a measured and metered quantity.

1Shamans were the first healers, and their heritage guides also many healers today. Shamanism exploits the innate capacity of the body to heal itself. Hippocrates and Galen, started as Asclepiads who adopted Shamanism in ancient Greece, and later founded the medical discipline regarded today as "scientific".

VMs allow CPU, memory, I/O, and network resources allocated to a database to be increased or decreased transparently. We can quickly suspend a live database application, reconfigure the database, and resume the application [1]. We can migrate a database live from one set of resources to another.

With all of these mechanisms in place, why do we still lack a truly self-healing database system? There seem to be two high-level reasons.

First, the space of possible failures that a truly self-healing database system has to deal with is huge. For example:

- There may be a hardware failure.
- The database configuration parameters or physical design may not be well tuned for the workload.
- Even if they are well tuned, the workload may change, sometimes drastically.
- An operator may configure the database.
- There may be a software bug.

Second, while there are many mechanisms readily available, there is a dearth of suitable policies to invoke these mechanisms automatically, efficiently, and correctly on failure. We need policies that detect failures in a timely fashion, find the right fix, and the right time to apply the fix.

A. Our Approach to Shaman

Shaman is a visionary project started at Duke with the aim of providing a holistic solution to self-healing. Shaman helps the system to recover to a healthy state by identifying the right set of failures and applying the requisite fixes rather than focusing on a specific failure/fix. However, building such a system is a challenging and time consuming task. Hence, Shaman takes an incremental approach to self-healing where we scope out a specific class of failures, and develop policies for self-healing in the face of these failures. This demonstration will present a part of larger vision of Shaman by focusing on the class of failures induced by workload changes in OLTP settings. Once matured, Shaman will support the mission-critical database applications consistently with good performance.

B. Shaman for workload changes in OLTP settings

Workload changes are a common cause of failures (in this case, performance problems) in database-backed Web services. Here, two types of changes can occur: (i) the overall query load changes (sometimes by 10x-100x), or (ii) the workload mix changes (e.g., a book starts to sell fast, causing more writes than usual).
The current way of dealing with this problem is overprovisioning (sometimes by 200-300%). Overprovisioning is a waste of resources and money for tight IT budgets. Moreover, overprovisioning is starting to become increasingly infeasible because data centers are running out of space and power. In fact, server consolidation is the trend in modern data centers.

To respond efficiently to failures caused by workload changes, Shaman needs to identify an appropriate fix quickly. Shaman addresses two challenging problems in this setting, that (to the best of our knowledge) have not been addressed before:

- There is a spectrum of different individual fixes: adjusting resources like CPU and memory, adjusting settings of configuration parameters like buffer pool sizes, or adjusting the physical design like indexes. In addition, the best fix may be some combination of these individual fixes.
- Should a proactive approach be taken to apply a given fix, or should it be applied reactively, or something in between? We demonstrate an interesting spectrum here where no one size fits all. As an example, VM technology enables CPU resources to be adjusted with low overhead, and these fixes act quickly. Thus, a reactive approach for CPU allocation may work fine. However, such an approach will not work for (costly) index creation in large databases, warranting a proactive approach. At the same time, to be proactive we need performance models and knowledge of workload patterns.

II. Overview of Shaman

Shaman currently works with DB2 and PostgreSQL. Suppose the (possibly well-tuned) database system is processing a workload $W_1$, and there is a workload change to $W_2$. If this workload change causes a performance problem—detected by Shaman through the violation of user-defined objectives on response time or throughput—then Shaman will try to fix the problem by applying a fix. There are two classes of fixes in Shaman:

- **Resource-based fixes** deal with allocation of physical resources like CPU, memory, I/O, and network. The current implementation of Shaman can dynamically reconfigure CPU and memory resources using the zones technology in Solaris. The same fix can be implemented using full-fledged VMs, but zones are more efficient because they are supported directly by the operating system.

- **Configuration-based fixes** include changes to the settings of configuration parameters as well as changes to the database physical design. Currently, Shaman considers buffer pool sizes and index (de)allocation.

When a problematic workload change happens, Shaman picks the least cost fix $F$ that will bring the database back to a healthy state. There are two important dimensions of cost: (A) How much time does $F$ take to bring the database back to a healthy state? and (B) How much extra resources had to be allocated? (In each case, the less the better.)

The summary of our empirical observations here is as follows:

- There is a spectrum of different individual fixes: adjusting resources like CPU and memory, adjusting settings of configuration parameters like buffer pool sizes, or adjusting the physical design like indexes. In addition, the best fix may be some combination of these individual fixes.
- Should a proactive approach be taken to apply a given fix, or should it be applied reactively, or something in between? We demonstrate an interesting spectrum here where no one size fits all. As an example, VM technology enables CPU resources to be adjusted with low overhead, and these fixes act quickly. Thus, a reactive approach for CPU allocation may work fine. However, such an approach will not work for (costly) index creation in large databases, warranting a proactive approach. At the same time, to be proactive we need performance models and knowledge of workload patterns.

III. Finding Robust Index Configurations

Many current database systems have deterministic query templates that administrators can expect to hit the database. For example, there are distinct query mixes for an OLTP system such as Amazon when users are browsing, selling, or purchasing books. Because the finite set of these templates is known ahead of time, Shaman can focus only on the relative mix of the workloads. The optimal index solution depends on the workload mix that hits the database at a given time. For example, a particular book may become popular, causing a dramatic spike in selling activity and making a specific type of update become dominant in the workload mix. In this case, new indexes may be created to provide high system performance.

Most current database systems can quickly produce an efficient index set $S$ given an input workload $W$ that has been recorded over time. However, it is impractical to assume that large databases will only have one type of workload—as previously mentioned, systems will often have varying, distinctly identifiable workloads throughout certain time periods. This section gives an overview of the following problem that Shaman considers: can we find one index set that is robust across the entire space of workload mixes that can hit the database system? If we can find such an index set, then we will not have to change the index set (especially create a new index) when a workload mix changes; this property is a big win due to the high cost of index creation.

Assume we are given a set of possible workload mixes $W_1, W_2, \ldots, W_l$. We can map each workload mix to a set of indexes recommended by an index advisor, such that $S_i = \text{Advise}(W_i)$. Given the optimal index set for each workload, we can calculate the optimal cost $c_i = \text{whatIf}(W_i, S_i)$, where whatIf is a function that returns the cost to execute a workload in the existence of a specific index set.

Note, that “optimal” here refers only to the optimal index set that the Advise tool could find in the time given to it. It may be possible to come across an index set $S_{i*}$ which leads to a cost $c_{i*} < c_i$. 

\[c_{i*} = \min_{S \in \text{Advise}(W_i)} \text{whatIf}(W_i, S)\]
two functions update/insert/delete statements). Therefore, the index set that increase the cost of updating indexes in write-queries (i.e., that adding indexes will decrease the read cost but may

subset of the node's parent. Assume the top node is in layer 0

Specifically, every node with an index set

As shown in Figure 1, we take the set union of all indexes in \( S_1, S_2, \ldots, S_n \) and obtain a candidate index space \( S = I_1, I_2, \ldots, I_m \). Now, we can define our problem as the selection of indexes from \( I_1, I_2, \ldots, I_m \) into our desired set \( S' \) such that the cost for each workload \( \text{whatIf}(W_i, S') \) achieves some sense of robustness. There are several different ways to define the robustness of the index set \( S' \). One requirement can be \( \text{whatIf}(W_i, S') \) should be under a certain threshold proportional to \( W_i \)'s optimal cost (e.g., \( 1.1 \times c_i \)) for \( i = 1, 2, \ldots, l \). This would guarantee that the execution of each workload with the index set \( S' \) does not perform much worse than its optimal cost. In some cases, administrators only care about a constant upper threshold for each workload, such as a number of minutes. A constant upper threshold \( t_i \) can be set for each workload \( W_i \), which requires \( \text{whatIf}(W_i, S') < t_i \).

A. Lattice-based Search with Pruning

The problem of finding a robust index set can be considered as a search problem in the lattice of possible index subsets of the candidate set. As shown in Figure 2, each subset of the entire index space, \( S \), can be modeled as a separate node in a complete lattice. If we put a node with the complete item set \( S \) at the top of the lattice, each node in the layer below is a subset of the node's parent. Assume the top node is in layer 0 with \( m \) indexes. Each node in layer \( k \) then has \( m - k \) indexes. Specifically, every node with an index set \( S_i \) is linked to it’s parents with a new index removed. This continues until the final node is \( \phi \).

The problem of index selection has the unique property that adding indexes will decrease the read cost but may increase the cost of updating indexes in write-queries (i.e., update/insert/delete statements). Therefore, the index set that is optimal for the union of the workloads is not robust, assuming that there is no disk-space constraint. Let us define two functions \( C_{\text{select}}(S_i) \) and \( C_{\text{write}}(S_i) \) for the aggregate costs of \( \text{SELECT} \) and \( \text{Write} \) respectively with index set \( S_i \) for all workloads. If the definition of robustness is an upper-bound \( c \), we can easily prune certain index sets. For an index set \( S_j \), where \( S_j \subseteq S_i \), if \( C_{\text{select}}(S_j) > t \), then we know that neither \( S_j \) nor \( S_i \) can be robust, as this original inequality implies \( C_{\text{select}}(S_j) > t \).

Based on the above observation, whenever \( C_{\text{select}}(S_i) > t \), we can recursively prune all child nodes of \( S_i \) from being the robust index set. This approach can be expanded by intelligently picking the nodes for the initial consideration. Specifically, at every iteration of the algorithm, the node with the least amount of robustness violations is chosen for the next iteration. If the algorithm begins with the parent node, this provides a quick pruning method.

In our empirical evaluations, we have observed that top nodes in the lattice often turn out to be robust—the intuition being that the benefits of indexes in reducing query execution costs outweigh by far the \( O(1) \) costs of updating indexes on updates/inserts/deletes—hence lattice-based search is a practical method.

B. Computing Benefits by Sampling Costs

We have also developed a different method to compute robust index sets. This method is based on sampling the costs of a carefully-selected small subset of index sets. From these sampled costs, we can compute a value as the benefit of each index. Interestingly, this value accounts for interactions among indexes (e.g., one index may be highly beneficial in the presence of another index, but useless otherwise). These benefit values can be used in different ways: (i) guiding the lattice-based search, or (ii) mapping index selection to knapsack formulations.

IV. Identifying Feasible Fixes

Recall that when the database encounters a performance problem, there may be a set of different fixes to resolve it. For instance, if the performance problem is caused by high volume of disk I/Os, two fixes could be applied: (i) adding more memory to improve the buffer cache hit ratio and thus reduce disk I/Os, or (ii) increasing disk bandwidth to accelerate disk accesses. This section describes how Shaman determines an effective and cost-efficient fix from the following candidates: (i) tuning the CPU resource, (ii) tuning the memory resource, and (iii) changing the configuration of buffer pool size. For each fix, we need to estimate its impact on performance and the cost associated with this fix. The overall workflow is shown in Figure 3.

A. Performance Models

To estimate the performance impact of a fix, we need to build a model for it. There are two types of models, i.e., black-box models such as classification and regression tree (CART) and white-box models such as queuing network.
models (QNsMs) [2]. Black-box models can be learned from the monitoring data without knowledge of the underlying system, so they are easy to adapt when the system changes. However, black-box models cannot capture the queuing effects that are common in database systems, which may lead to poor accuracy in predicting system performance. A QNM is composed of mathematical formulas that are devised to simulate the interactions between the workload and system resources. With appropriate parametrization, a QNM can be used to predict performance metrics of interest by solving the model. Since QNMs leverage the domain knowledge of the underlying system when modeling the queuing effects, they are expected to outperform black-box models at the cost of adaptation due to system change. Shaman adopts a mix of queuing network models and black-box models to balance prediction accuracy and model adaptability.

We use a closed-queuing network model to describe the whole system. Suppose there are \( n \) users in the system, CPU and disks are two types of service centers in the system. A user request needs a certain amount of processing time (also called service demand) at each service center. Between two consecutive requests, a user spends some think time \( z \).

Given a queuing network model \( Q \) with parameters \( n, z \), and service demand \( d_i \) per request at \( i \)th service center, we can apply mean value analysis (MVA) to solve \( Q \) and predict the system performance \( p \). The overall model is expressed as \( p=Q(d_1, \ldots, d_k, n, z) \), where \( k \) is the number of service centers in the system. In this model, \( p, n, \) and \( z \) can be observed from the system, while service demands \( d_1, \ldots, d_k \) depend on the workload \( W \), CPU resource \( r_c \), memory resource \( r_m \), and buffer pool size \( b \). We use CARTs to capture such relationships as \( d_i = F_i(W, r_c, r_m, b) \).

**B. Training Performance and Cost Models**

We can run a bunch of experiments by varying the setting of workload \( W \), CPU resource \( r_c \), memory resource \( r_m \), and buffer pool size \( b \) of a Web service system. When experiments are running, we record the performance value \( p \) (e.g., average response time or throughput), average CPU utilization, and average disk utilization for each monitoring interval.

Service demand for both service centers (i.e., CPU and disk) is estimated using utilization law [2]:

\[
Demand = Utilization \times Time / completions,
\]

where Demand is the average service time that each request in a workload gets from the service center; Utilization is the percentage of time that a service center is busy with processing requests; Time is a monitoring interval; and Completion is the total amount of completed requests in the monitoring interval.

Thus we can train a black-box model to estimate service demand \( d_i \) based on the setting of \( (W, r_c, r_m, b) \). The queuing network model \( Q \) is calibrated to predict the overall system performance. With a mix of queuing network model \( Q \) and black-box models \( F_i(1 \leq i \leq k) \), we can predict performance impact of a hypothetical change of \( r_c, r_m, \) and \( b \) under a workload mix \( W \).

Recall from Section II that each fix has one or more associated costs. Models to estimate these costs are learned in the similar way as performance models. In the initial version of Shaman, both performance models and cost models are learned from data generated offline.

**C. Picking the Least Cost Fix**

As shown in Figure 3, when a performance problem is detected (i.e., the performance value does not meet a required value \( p_0 \)), a list of candidate fixes is generated using the performance model and we pick a feasible fix with the least cost to bring the system back into normal states. Picking the least cost fix is essentially a constrained optimization problem:

\[
\min_{r_c, r_m, b} \{ \text{Cost}(r_c, r_m, b) \}
\]

such that

\[
Q(d_1, \ldots, d_k, n, z) < p_0
\]

and

\[
d_i = F_i(W, r_c, r_m, b); (1 \leq i \leq k)
\]

**V. Demonstration Plan**

Our demonstration will use two open-source Web services: Rubis, the eBay-like auction service, and TPC-W, the Amazon-like book-selling service. We have workload generators for both these services that will be used to induce performance problems through workload changes.

**A. Shaman in Action**

The first session of our demonstration will focus on showing how Shaman works. We will show how Shaman:

- Finds robust index sets.
- Quickly gets the database back into a healthy state through CPU, memory, and buffer-pool fixes; and their combinations.
- Dynamically adjusts how it costs resources based on which resource is more in demand now.

**B. Shaman Vs. A Purely Reactive Approach**

The second session of our demonstration will show three case studies that bring out the disadvantages of a purely reactive approach (what a database administrator would do today using online problem diagnosis) compared to Shaman:

- Reactive cannot identify the right fix.
- Reactive cannot identify the magnitude of the right fix to apply (e.g., should CPU be increased by 10% or 50%?).
- Reactive picks a costlier fix when multiple fixes will work.

**C. Insights**

The final session will bring out some of the insights from Shaman, e.g., comparison of the use of black-box Vs. white-box models in Shaman based on accuracy of fix identification, ability for extrapolation, and training time.

**REFERENCES**