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

I. 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 [6]. 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*[1], 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 were developed because of the decreasing tolerance towards service downtime.[2] Examples of such mechanisms include:

- 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 most database systems required a database restart; but not any more.
- Many database systems can now dynamically assimilate new hardware resources without any service downtime. For example, Microsoft SQL Server provides "Hot-add CPU" and "Hot-add memory" features.
- Many database systems can now run administrative tasks in a throttled mode without any unbounded effect on the production workload. With this feature, resource-intensive tasks like backups, statistics update, and table/index defragmentation 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. VMs allow CPU, memory, and I/O 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 [2]. We can migrate a database live from one set of resources to another.

[1]Shamans were the first healers, and their heritage guides 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".

[2]Brokerages and banking firms can lose up to $75,000 per minute of downtime. A 22-hour outage at eBay cost the company more than $3 Million in customer credits and $4 Billion in market capitalization.

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 misconfigure 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. Shaman's Approach to Self-Healing

Shaman is a long-term project that will explore mechanisms and policies for self-healing database systems. We are taking an incremental approach to deal with this complex problem. Our approach involves scoping out specific classes of failures, and developing mechanisms and polices for self-healing in the face of such failures. Over time, we will combine the individual solutions to support mission-critical database applications with consistent good performance under larger and larger spaces of potential failures. Our current demonstration will present a part of Shaman's ambitious vision by focusing on the class of failures induced by workload changes in OLTP settings. We will demonstrate Version 1.0 of Shaman.

B. Shaman 1.0: Dealing with OLTP Workload Changes

Workload changes are a common cause of failure (in this case, performance problems) in database-backed Web services. Two types of workload changes can occur: (i) the overall load changes (sometimes by 10x-100x), or (ii) the request mix changes (e.g., a book starts to sell fast, causing more writes than usual). The current way to deal with this problem is overprovisioning (sometimes by 200-300%). Overprovisioning wastes resources and money in tight IT budgets. Moreover, overprovisioning is becoming 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.

To the best of our knowledge, these two problems have not been addressed previously:

- 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 (possibly) knowledge of workload patterns.

II. **Overview of Shaman**

Shaman currently works with IBM 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 1.0:

- *Resource-based fixes* deal with allocation of physical resources like CPU, memory, and I/O bandwidth. The current implementation of Shaman can dynamically reconfigure CPU and memory resources using the *zones* technology in Solaris [5]. 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:

- Index creation is a very costly fix, so it is better (if possible) to avoid getting into a state where this fix needs to be applied. Interestingly, Section III will show that such a proactive solution is possible.
- The CPU, memory, and buffer pool fixes are tuning knobs that can be adjusted reactively; but we need models that can (roughly) estimate by how much to turn the knobs when a workload change happens (Section IV).

III. **Finding Robust Index Configurations**

Workloads in many database systems are generated by deterministic SQL templates that database administrators know well in advance. The workload in a database system used by an e-commerce application or a report-generation application invariably consists of a mix of SQL statements generated by SQL DML templates embedded in the application. For example, in an e-commerce setting like Amazon.com, there are distinct templates involved in supporting user interactions like browsing, searching, purchasing, or selling books. The optimal set of indexes that a database system needs at any given point of time depends on the workload at that time. If the workload changes, then the optimal index set may also change [1]. For example, a particular book sold by Amazon.com may suddenly become popular, causing a dramatic spike in ordering activity; making some specific SQL insert and update statements dominant in the workload. In this setting, new indexes may need to be created to maintain good system performance.

Current database systems can identify an efficient index set given an input workload. However, it is impractical to assume that enterprise databases have one type of workload always: these systems usually have different, distinctly-identifiable workloads through different 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 workloads that can hit a database system in an OLTP setting? If such an index set is found, then Shaman does not have to change the index configuration (especially, create new indexes) when the workload changes. Robust indexes can be a big win due to the high cost of index creation in a database system under production use (which we will demonstrate).

Suppose we are given a set of possible workloads $W_1, W_2, \ldots, W_l$. Each workload W_i can be mapped to a set of indexes S_i recommended by a conventional index advisor (e.g., DB2's *db2advis*). Given the chosen index set for each workload, we can calculate the *optimal cost* $c_i = \texttt{whatIf}(W_i, S_i)$, where whatIf is a function that returns the cost to execute a workload (W_i) when a specific index set (S_i) is present in the database.[3]

As shown in Figure 1, Shaman takes the set union of all indexes in $S_1, S_2, \ldots, S_l$ to obtain a *candidate index space* $S = I_1, I_2, \ldots, I_m$.[4] We can now define our problem as the selection of indexes from $I_1, I_2, \ldots, I_m$ into a single set S^* such that the cost of S^* for each workload achieves some notion of robustness. There are several ways to define robustness, and the appropriate definition depends on application needs. One sufficient requirement can be that $\texttt{whatIf}(W_i, S^*)$, for $i = 1, \ldots, l$, is lower than a certain threshold proportional to W_i's optimal cost. For example, $\texttt{whatIf}(W_i, S^*) \leq (1+\alpha)c_i$, for some small positive α like 0.1. This requirement ensures that

[3]Note that "optimal" here refers to the best index set that the index advisor could find in the time given to it. It may be possible to find an index set S_{i*} that leads to a cost $\texttt{whatIf}(W_i, S_{i*}) < c_i$.

[4]It is possible to adjust the size of the candidate space using techniques like index merging [3].

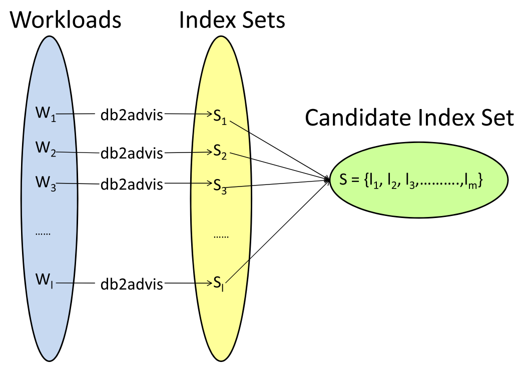

Fig. 1. Identifying the candidate index set

the cost of execution of each workload W_i with the (robust) index set S^* is not much worse than W_i's optimal cost.

Alternately, administrators may only care about a fixed cost threshold T for all workloads, e.g., a maximum tolerable average response time. Thus, S^* is robust if $\texttt{whatIf}(W_i, S^*) \leq T$ for $i = 1, \ldots, l$. T could also vary across different W_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 candidate index space S forms a distinct node in the lattice. The node with the complete index set is at the top of the lattice. Assume the topmost node is in Layer 0 and has all m candidate indexes. Then, each node S' in Layer k has $m-k$ indexes. S' is linked to its children at Layer $k+1$, where each child has one index removed from S'. This structure continues until the final node is ϕ in Layer m.

We can write $\texttt{whatIf}(W_i, S') = C_{select}(W_i, S') + C_{other}(W_i, S')$. $C_{select}(W_i, S')$ is the cost of the read-only part of Workload W_i, e.g., the cost of the SQL *Select* statements in W_i. The robust index selection problem has the interesting property that adding new indexes to an existing set S' will not increase $C_{select}(W_i, S')$. (Intuitively, as we go up the lattice in Figure 2, C_{select} costs tend to drop.) This property can be used to *prune* large parts of the lattice without exploration. For example, let the definition of robustness be based on a fixed threshold T. Now suppose we find an index set S' such that $C_{select}(W_i, S') > T$. Then, any index set $S'' \subseteq S'$ will not be robust because it is guaranteed that $C_{select}(W_i, S'') > T$. Thus, S' and all its descendants in the lattice can be pruned if $C_{select}(W_i, S') > T$.

We have extended the above intuition into a complete algorithm, called *prioritized-prune search*, that tries to explore the lattice top-down in a smart fashion. Exploring a node S' involves computing $\texttt{whatIf}(W_i, S')$ costs, $i = 1, \ldots, l$, for S' as well as all immediate children of S'. The topmost node is explored first. Successive nodes are chosen for exploration based on a notion of priority that depends on the number of violations of robustness seen at the unexplored children of each explored node. Pruning is used to reduce unfruitful node explorations.

Our empirical evaluations show that one or more higher-up nodes in the lattice often turn out to be robust in OLTP settings. The intuition here is that the benefits of indexes in

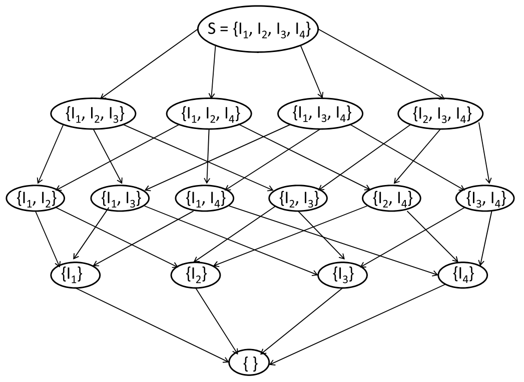

Fig. 2. Index Lattice

reducing query execution costs outweigh the $O(1)$ costs of updating indexes on inserts/deletes/updates. Hence, prioritized-prune search, while not particularly elegant, usually works in practice.

B. Computing Benefits by Sampling Costs

We have also developed a more scalable algorithm to find robust index sets. This method is based on sampling the costs of a carefully-chosen small subset of index sets from the lattice. From these sampled costs, Shaman computes values B_{ij} that represent the *benefit* of each index I_j for each workload W_i. Interestingly, these values account for *interactions* among different indexes (e.g., one index may be highly beneficial in the presence of another index, but useless otherwise). Our algorithm uses these benefit values in two different ways: (i) mapping robust index selection to a knapsack formulation, and (ii) guiding the prioritized search of the lattice.

IV. Identifying Feasible Fixes

When the database encounters a performance problem, there may be different potential fixes to resolve the problem. For instance, if the performance problem is caused by a high volume of disk I/Os, two potential fixes apply: (i) increasing the memory allocation to the database to improve the buffer pool hit ratio, or (ii) increasing disk bandwidth to accelerate I/Os. 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) tuning the database buffer pool size. We assume that a robust index configuration is available.

For each fix, Shaman needs to estimate its impact on database performance and also the cost associated with this fix. The overall workflow is shown in Figure 3; described next.

A. Performance Models

Two types of models are applicable to estimate the performance impact of various fixes: black-box models such as *classification and regression trees* (CARTs) and white-box models such as *queuing network models (QNMs)* [4]. Black-box models can be learned from database monitoring data without knowledge of the underlying system internals; so they are easy to adapt when significant system changes occur. However, black-box models cannot capture the queuing effects present in OLTP systems, which may lead to poor accuracy in predicting performance. A QNM is composed of mathematical formula that simulate the interactions between the workload and system resources. With appropriate parametrization, a QNM can be used to predict performance metrics of interest. QNMs leverage domain knowledge about the system when modeling queuing effects, so QNMs are expected to outperform black-box models in prediction accuracy; at the cost of reduced adaptability to system changes.

To balance prediction accuracy and model adaptability, Shaman uses a hybrid approach that combines QNMs and black-box models. Since OLTP settings are the current target, a closed QNM is used to represent the database system. Let n be the number of concurrent users in the system. System resources like CPU and disks form *service centers* in the QNM. An input request composed of SQL statements needs a certain amount of processing time (called *service demand*) at each service center. Between two consecutive requests, a user spends some *think time* z on average.

Given a QNM Q with parameters n, z, and service demand d_i per request at the i^{th} service center, we can apply *mean value analysis (MVA)* to solve Q and predict the performance p [4]. The overall model has the form $p{=}Q(d_1, \ldots, d_k, n, z)$, where k is the number of service centers in the system. In this model, p, n, and z can be observed through system monitoring, 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. (Currently, all factors apart from W, r_c, r_m, and b are assumed to be fixed.) We use CARTs to capture such relationships in the form $d_i = F_i(W, r_c, r_m, b)$.

B. Training the Performance and Cost Models

In the current version of Shaman, both performance and cost models are learned from data generated through offline *experiments*. (Policies and mechanisms for online experiments are under development.) Each experiment considers a chosen setting of the workload W, CPU resource r_c, memory resource r_m, and buffer pool size b; and produces the performance p for that setting. p includes, e.g., average response time or throughput, resource utilization, and other metrics of interest.

The service demand of a service center is estimated using the *utilization law* for QNMs [4]: *Demand = Utilization*$\times$*Time*/*Completions*, where *Demand* is the average time that each request in the workload spends in the service center; *Utilization* is the fraction of time that the service center is busy processing requests; and *Completions* is the total number of requests completed in the monitoring interval *Time*. The data collected through experiments is used to train the black-box model F_i to estimate service demand d_i based on the setting of $\langle W, r_c, r_m, b\rangle$. Models to estimate the costs of fixes (recall Section II) are learned in a similar fashion.

C. Picking the Least Cost Fix

As shown in Figure 3, when a performance problem is detected—e.g., the average response time exceeds a threshold p_0—Shaman first uses the performance models to generate a set of *feasible fixes* (fixes that can solve the problem and bring the system back to a healthy state). The cost models are then used to find the least-cost feasible fix. In effect, Shaman solves the following constrained optimization problem:

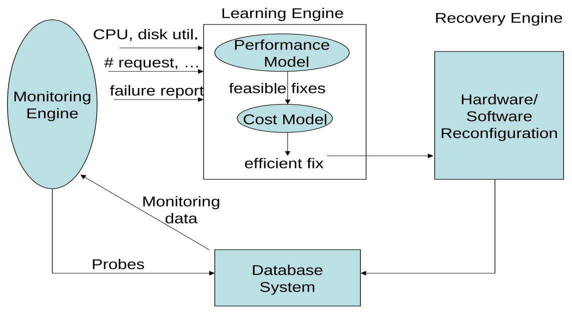

Fig. 3. Shaman's workflow for self-healing

$$\min_{r_c, r_m, b} \{\texttt{Cost}(r_c, r_m, b)\} \;\; \textit{such that}$$
$$Q(d_1, \ldots, d_k, n, z) \leq p_0, \;\textit{and}\; d_i{=}F_i(W, r_c, r_m, b), i \in [1, k]$$

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. Workload generators for these services 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 brings the database back to a healthy state through the appropriate CPU, memory, and buffer-pool fixes.
- Dynamically adjusts how it costs resources based on which resource is in more 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 based on the accuracy of fix identification, ability for extrapolation, and training time.

References

[1] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[2] B. Chandramouli, C. Bond, S. Babu, and J. Yang. On suspending and resuming queries. In *SIGMOD*, 2007.

[3] S. Chaudhuri and V. R. Narasayya. Index merging. In *ICDE*, 1999.

[4] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[5] *Dtrace, ZFS, and Zones in Solaris*. www.sun.com/software/solaris.

[6] D. Zhao. An evaluation of techniques for self-healing in application and database servers. M.S. thesis, Duke University, 2008.